LIHORNE.COM

CS 245
Logic & Computation

Dr. Borzoo Bonakdarpour • Fall 2013 • University of Waterloo

Last Revision: February 13, 2014

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. If you spot any errors or would like to contribute, please contact me directly.

1 Introduction

The Neehdham-Shroeder Authentication Protocol is a security protocol that works like this. Suppose there are two people, Alice and Bob. Alice intends to send a message to Bob, and this message can be represented by $\{A, N_a\}_{PK_B}$. Alice wants to establish secure communication with Bob. The idea is that a message can be encrypted by a public key (e.g., PK_B (public key of Bob)) that only Bob can decrypt using his private key. The returned message is now $\{N_a, N_b\}_{PK_A}$. Alice authenticates Bob. Bob then authenticates Alice, returning $\{N_b\}_{PK_B}$.

The problem with this was discovered in 1997, using program verification. Suppose there is an intruder in the middle, and he convinces Alice to send her message encrypted using the Intruder's public key. Then, the intruder sends the message encrypted with Bob's public key back to Bob, and Bob responds with Alice's message encrypted with her public key. He then sends it back to Alice, then she replied with the Intruder's public key and again he can decrypt it and send it back to Bob. Through these means, the intruder successfully impersonates Alice.

1.1 Background

Definition 1.1 (set)**.** A **set** is a collection of objects called **members** or **elements**. We write

$$\alpha \in S$$

to mean that α is a member of S ($\alpha \notin S$ is the opposite). We write

$$\alpha_1, \cdots, \alpha_n \in S$$

to mean that $\alpha_1 \in S, \cdots$, and $\alpha_n \in S$.

Two sets are **equal** if and only iff they have the same members. That is,

$$\text{for every } x, x \in S \iff x \in T$$

S is said to be a **subset** of T, written as

$$S \subseteq T$$

iff for every $x, x \in S$ implies $x \in T$. Every set is a subset of itself. $S = T$ iff $S \subseteq T$ and $T \subseteq S$.
S is a **proper subset** of T ($S \subset T$) if and only if $S \subseteq T$ and $S \neq T$. Sets are not ordered (for example, $\{\alpha, \beta\} = \{\beta, \alpha\}$). Additionally, the empty set is denoted $\emptyset$ and is a set which has no members at all.

Definition 1.2 (complement, union, intersection, difference)**.** $\overline{S} = \{x | x \notin S\}$ is the complement.
$S \cup T = x | x \in S$ or $x \in T$ is the union.
$S \cap T = x | x \in S$ and $x \in T$ is the intersection.
$S - T = x | x \in S$ and $x \notin T$ is the difference.

S and T are said to be **disjoint** iff $S \cap T = \emptyset$.

Definition 1.3 (union)**.** The **union** of $\{S_i | i \in I\}$ is defined by

$$\bigcup_{i\in I} S_i = \{x | x \in S_i \text{ for some } i \in I\}$$

Definition 1.4 (intersection)**.** The **intersection** of $\{S_i | i \in I\}$ is defined by

$$\bigcap_{i\in I} S_i = \{x | x \in S_i \text{ for each } i \in I\}$$

Definition 1.5. (natural numbers)

1. $0 \in \mathbb{N}$
2. For any n, if $n \in \mathbb{N}$, then $n' \in \mathbb{N}$, where n' is the successor of n.
3. $n \in \mathbb{N}$ only if n has been generated by [1] and [2].

Example 1.1. Show that

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Proof. Base case: $n = 2$.
Inductive step: We assume that

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

We now show that

$$1 + 2 + \cdots + k + 1 = \frac{(k+1)(k+2)}{2}$$

□

2 Propositional Logic

Definition 2.1 (logic)**. Logic** is the science of principles of valid reasoning and inference. The aim of logic in computer science is to develop languages to model the situations we encounter, so that we can **reason** about them formally. **Reasoning** about situations means constructing arguments about them. We want these arguments to be formal, can be defended rigorously, or executed on a machine.
In propositional logic, **simple(atomic)** propositions are the basic building blocks used to create **compound** propositions using connectives.

We will construct a **propositional language** $\mathcal{L}^p$ and it is the formal language for propositional logic. A formal language is a collection of symbols, distinguished from symbols of the metalanguage used in studying them.
$\mathcal{L}^p$ consists of three classes of symbols

1. propositional symbols we use roman-type small Latin letters (e.g., p q r). The set of propositional symbols is denoted by $Atom(\mathcal{L}^p)$.
2. Five connective symbols/connectives (negation, conjunction, disjuction, implication, equivalence)

$$\neg, \wedge, \vee, \rightarrow, \leftrightarrow$$

3. Punctuation; we use (and)

Definition 2.2 (expression)**. Expressions** are finite strings of symbols. The expression of length 0 is called the empty expression which cannot be written. We use the notation $\emptyset$ to denote the empty expression. The **length** of an expression is the number of occurences in it. Two expressions are equal if they have the same length and have the same symbols in order.

Definition 2.3 (segment)**.**

Consider two expressions U and V in this order, then their concatenation is UV. If some expression $U = W_1VW_2$ then V is a segment of U; if $U \neq V$, then V is a proper segment of U. If $U = VW$, then V is an **initial** segment of U and W is a **terminal** segment of U. If W is non-empty, then V is a **proper initial segment** and if V is non-empty then W is a proper terminal segment.

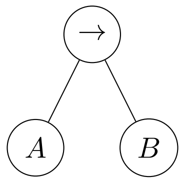
Definition 2.4 (formula)**.** formulas are defined from expressions. The set of **formulas** of $\mathcal{L}^p$ (denoted $Form(\mathcal{L}^p)$) is inductively defined as follows:

1. $Atom(\mathcal{L}^p) \subseteq Form(\mathcal{L}^p)$
2. If $A \in Form(\mathcal{L}^p)$, then $(\neg A) \in Form(\mathcal{L}^p)$
3. If $A, B, \in Form(\mathcal{L}^p)$, then $(A * B) \in Form(\mathcal{L}^p)$, where $*$ is a binary connective.

We indicate roman capital letters to indicate formulas, such as $A, B, C, U, V, etc.$ Note also that $Form(\mathcal{L}^p)$ is the smllest class of expression of $\mathcal{L}^p$ **closed** under the formation rules of $\mathcal{L}^p$.
There are certain formula types:

- $(\neg A)$ is called a negation
- $(A \wedge B)$ is called a conjunction
- $(A \vee B)$ is called a disjunction
- $(A \to B)$ is called an implication
- $(A \leftrightarrow B)$ is called an equivalence

We can build **parse trees**. For example the expression $(A \to B)$ has the parse tree



Another method is through the use of a recursive algorithm like this.

- **Input:** U is an expression of $\mathcal{L}^p$
- **Output: true** if U is in $Form(\mathcal{L}^p)$; **false** otherwise
- **Steps:**
  1. Return **false** if the formula is empty.
  2. If $U \in Atom(\mathcal{L}^p)$, then return **true**; otherwise if U is any other single symbol, return **false**
  3. If U contains more than one symbol and it does not start with '(', then return **false**

4. If the second symbol is $\neg$, U must be $(\neg V)$ where V is an expression; otherwise return **false**. Now, recursively apply the same algorithm to V, which is of smaller size.
5. If U begins with '(' but the second symbol is not $\neg$, scan from left to right until(V segment is found where V is a proper expression; if no such V is found, return **false**. U must be $(V * W)$ where $W$is also an expression; otherwise return **false**.
6. Now apply the same algorithm recursively to V and W.

Since every expression is finite in length by definition, and since in each iteration the analyzed expressions are getting smaller, the algorithm terminates in a finite number of steps.
Question. How should we prove that every formula has the equal number of left and right parentheses?

Remark 2.1. In **course-of-values induction** the induction hypothesis for proving $M(n+1)$ is not just $M(n)$, but the conjunction

$$M(1) \wedge M(2) \wedge \cdots \wedge M(n)$$

Thus, there does not have to be an explicit induction base case.

In order to prove properties of propositional formulas, we apply induction on the height of the parse tree. This proof is called structural induction.

Lemma 2.1. Every formula $\mathcal{L}^p$ has the same number of left and right parentheses.

Proof. Let $M(n)$ mean 'All formulas A of height n that have the same number of left and right brackets'. We assume $M(k)$ for each $k < n$ and try to prove $M(n)$. The base case is that $n = 1$. This implies that $A \in Atom(\mathcal{L}^p)$ and hence has 0 parentheses. Next, our inductive step for $n > 1$. The root of the parse tree φ must be in the connectives. Without loss of generality we can assume that it is $\rightarrow$ and $\varphi = (\varphi_1 \rightarrow \varphi_2)$. The heights of φ_1 and φ_2 have to be strictly less than n. Using the inductive hypothesis, the nmber of left and right parentheses in φ should also be equal because we simply added 2 more parentheses. □

Lemma 2.2. Any non-empty proper initial segmentof a formula of $\mathcal{L}^p$ has more left than right parentheses, and any non-empty proper terminal segment of a formula of $\mathcal{L}^p$ has less left than right parentheses.

Theorem 2.1 (formulas uniqueness)**.** Every formula of $\mathcal{L}^p$ is of exactly one of six forms: an atom, $(\neg A), (A \wedge B), (A \vee B), (A \rightarrow B)$, and $(A \leftrightarrow B)$; and in each case it is of that form in exactly one way.

We now begin to examine the scope of a formula. We begin with a definition for scope.

Definition 2.5 (scope)**.** If $(\neg A)$ is a segment of C, then A is called the **scope** in C of the $\neg$ on the left of A. If $(A * B)$ is a segment of C, then A and B are called the left and right **scopes** in C of the $*$ between A and B.

Theorem 2.2 (unique scope)**.** Any $\neg$ in any A has a unique scope, and any $*$ in any A has unique left and right scope.

Theorem 2.3. If A is a segment of $(\neg B)$ then A is a segment of B or $A = (\neg B)$. If A is a segment of $(B * C)$ then A is a segment of B, or A is a segment of C, or $A = (B * C)$.

2.1 Semantics of Propositional Logic

Definition 2.6 (semantics)**.** Informally, **semantics** of a logic describe how to interpret formulas. For example, the interpretation of formula $p \wedge q$ depends on three things: the interpretation of p, the interpretation of q, and the interpretation of $\wedge$. In propositional logic, we need to give **meaning** to atoms, connectives, and formula.

Let A and B be two formulas that express propositions $\mathcal{A}$ and $\mathcal{B}$. Intuitively we give the following meanings:

$\neg A$	Not $\mathcal{A}$
$A \wedge B$	$\mathcal{A}$ and $\mathcal{B}$
$A \vee B$	$\mathcal{A}$ or $\mathcal{B}$
$A \rightarrow B$	If $\mathcal{A}$ then $\mathcal{B}$
$A \leftrightarrow B$	$\mathcal{A}$ iff $\mathcal{B}$

Definition 2.7 (semantics)**.** Formally, semantics is a function that maps a formula to a value in $\{0, 1\}$ (also known as a **truth table**)

A	¬ A
1	0
0	1

A **truth valuation** is a function with the set of all proposition symbols as domain and $\{0, 1\}$ as range. Note that $1 \rightarrow 1$ is 1 because truth is preserved, $1 \rightarrow 0$ is 0 because truth is not preserved, and $0 \rightarrow 0$ is 1 because there is no truth to be preserved.

Definition 2.8 (value)**.** The **value** assigned to formulas by a truth valuation t is defined by recursion:

[1] $p^t \in \{0, 1\}$

[2] $(\neg A)^t = \begin{cases} 1 & \text{if } A^t = 0 \\ 0 & \text{if } A^t = 1 \end{cases}$

[3] $(A \wedge B)^t = \begin{cases} 1 & \text{if } A^t = B^t = 1 \\ 0 & \text{otherwise} \end{cases}$

[4] $(A \vee B)^t = \begin{cases} 1 & \text{if } A^t = 1 \text{ or } B^t = 1 \\ 0 & \text{otherwise} \end{cases}$

[5] $(A \rightarrow B)^t = \begin{cases} 1 & \text{if } A^t = 0 \text{ or } B^t = 1 \\ 0 & \text{otherwise} \end{cases}$

[6] $(A \leftrightarrow B)^t = \begin{cases} 1 & \text{if } A^t = B^t \\ 0 & \text{otherwise} \end{cases}$

Definition 2.9 (compositional)**.** Notice that semantics of propositional logic is **compositional**; i.e., if we know the valuation of two subformulas, then we know the valution of their composition using a propositional connective.

An easy approach for evaluating propositional formulas is by building truth tables by considering all combinations. In general, for n propositional variables, there exist 2^n values.

Suppose $A = p \vee q \rightarrow q \wedge r$. If $p^t = q^t = r^t = 1$, then $A^t = 1$ Also, if $p^{t_1} = q^{t_1} = r^{t_1} = 0$, then $A^{t_1} = 1$.

Theorem 2.4. For any $A \in Form(\mathcal{L}^p)$ and any truth valuation t, $A^t \in \{0, 1\}$.

Definition 2.10 (satisfiable)**.** Let Σ denote a set of formulas and

$$\Sigma^t = \begin{cases} 1 & \text{if for each } B \in \Sigma, B^t = 1 \\ 0 & \text{otherwise} \end{cases}$$

We say that Σ is **satisfiable** iff there is some truth valuation t such that $\Sigma^t = 1$. When $\Sigma^t = 1$, t is said to **satisfy** Σ.

For example, the set $\{(p \to q) \vee r, (p \vee q \vee s)\}$ is satisfiable.

Definition 2.11 (tautology)**.** A formula A is a **tautology** iff for any truth valuation t, $A^t = 1$.

Definition 2.12 (contradiction)**.** A formula A is a **contradiction** iff for any truth valuation t, $A^t = 0$.

A faster way to evaluate a propositional formula is by using valuation trees and "expressions".

Example 2.1. Show that $A = ((((p \wedge q) \to r) \wedge (p \to q)) \to (p \to r))$ is a tautology.

Definition 2.13 (deducible)**.** Suppose $\mathcal{A}_1, \cdots, \mathcal{A}_n$, and $\mathcal{A}$ are propositions. Deductive logic studies whether $\mathcal{A}$ is **deducible** from $\mathcal{A}_1, \cdots, \mathcal{A}_n$.

Definition 2.14 (tautological consequence)**.** Suppose $\Sigma \subseteq Form(\mathcal{L}^p)$ and $A \in Form(\mathcal{L}^p)$. We say that A is a **tautological consequence** of Σ (that is, of the formulas in Σ), written as $\Sigma \models A$, iff for any truth valuation t, $\Sigma^t = 1$ implies $A^t = 1$. Note that $\Sigma \models A$ is not a formula.

We write $\Sigma \not\models A$ for "not $\Sigma \models A$". That is, there exists some truth valuation t such that $\Sigma^t = 1$ and $A^t = 0$. Also, $\emptyset \models A$ means that A is a tautology.

Example 2.2. $\{(A \to B), (B \to C)\} \models A \to C$

Example 2.3.

$$\{((A \to \neg B) \vee C), (B \wedge (\neg C)), (A \leftrightarrow C)\} \not\models (A \wedge (B \to C)).$$

Definition 2.15 (associativity of commutativity)**.**

$$\begin{gathered}(A \wedge B) \equiv (B \wedge A)\\ ((A \wedge B) \wedge C) \equiv (A \wedge (B \wedge C))\\ (A \vee B) \equiv (B \vee A)\\ ((A \vee B) \vee C) \equiv (A \vee (B \vee C))\end{gathered}$$

Theorem 2.5.

[1] $\{A_1, \ldots, A_n\} \models A \iff \emptyset \models A_1 \wedge \cdots \wedge A_n \to A$

[2] $\{A_1, \ldots, A_n\} \models A \iff \emptyset \models A_1 \to (\cdots (A_n \to A) \cdots)$

Lemma 2.3. If $A \equiv A'$ and $B \equiv B'$ then,

1. $\neg A \equiv \neg A'$
2. $A \wedge B \equiv A' \wedge B'$
3. $A \vee B \equiv A' \vee B'$
4. $A \to B \equiv A' \to A' \to B'$
5. $A \leftrightarrow B \equiv A' \equiv B'$

Theorem 2.6 (replaceability)**.** If $B \equiv C$ and A' results from A by replacing some (not necessarily all) occurrences of B in A by C, then $A \equiv A'$.

Proof. By induction on the structure of A. If $B = A$, then $C = A'$. This theorem thus holds. Basis. A is an atom. Then $B = A$; the theorem holds. Induction step. A is one of the five forms: $\neg A_1$, $A_1 \wedge A_2$, $A_1 \vee A_2$, $A \to A_2$, $A_1 \leftrightarrow A_2$.
Suppose $A = \neg A_1$. If $B = A$, the theorem holds as stated above. If $B \not{=} A$, then B is a segment of A_1. Let A_1' results from A_1 by the replacement stated in the theorem, then $A' = \neg A_1'$. We have

$$A_1 \equiv A_1' \quad \text{(by inductive hypothesis)},$$

$$\neg A_1 \equiv \neg A_1'$$

That is, $A \equiv A'$. Suppose $A = A_1 * A_2$. ($*$ denotes any one of $\wedge, \vee, \to, \leftrightarrow$.) If $B = A$, the theorem holds as in the above case. If $B \not\equiv A$, then B is a segment of A_1 or A_2 (by Theorem 2.3.7). Let A_1' and A_2' result respectively from A_1 and A_2 by the replacement stated in the theorem, then $A' = A_1' * A_2'$. We have

$$A_1 \equiv A_1', A_2 \equiv A_2' \quad \text{(by inductive hypothesis)}$$

$$A_1 * A_2 \equiv A_1' * A_2'$$

That is, $A \equiv A'$. By the basis and induction step, the theorem is proved. □

Theorem 2.7 (duality)**.** Suppose A is a formula composed of atoms and the connectives $\neg$, $\wedge$, and $\vee$ by the formation rules concerned, and A' results by exchanging in A, $\wedge$ for $\vee$ and each atom for its negation. Then $A' \equiv \neg A$. (A' is the **dual** of A)

Formulas $A \to B$ and $\neg A \vee B$ are tautologically equivalent. Then $\to$ is said to be **definable** in terms of (or **reducible**) $\neg$ and $\vee$.

Let f and g be two n-ary connectives. We shall write $fA_1, \dots, A_n$ for the formula formed by an n-ary connective f connectiving formulas $A_1, \dots, A_n$.
Question. Given $n \geq 1$, how many n-ary connectives exist?

Example 2.4. Suppose f_1, f_2, and f_3 are distinct unary connectives. They have the following truth tables:

A	$f_1 A$	$f_2 A$	$f_3 A$	$f_4 A$
1	1	1	0	0
0	1	0	1	0

Definition 2.16 (adequate)**.** A set of connectives is said to be **adequate** iff any n-ary ($n \geq 1$) connective can be defined in terms of them.

Theorem 2.8. $\{\wedge, \vee, \neg\}$ is an adequate set of connectives.

Corollary 2.1. $\{\wedge, \neg\}, \{\vee, \neg\}, \{\to, \neg\}$ are adequate.

2.2 Proof Systems in Propositional Logic

We would like to construct a **calculus** for reasing about propositional logic. The application of such a calculus is to mechanize proofs of validity. To mechanically develop proofs, we need **proof rules**. Using a proof rule, one can **infer** a formula from another.

Definition 2.17 (Hilbert System)**.** The **Hilbert System** (H) is an example of a deduction system for the set of propositional logic formulas. A well-formed formula A is formally **provable** by Hilbert System axioms H if and only if

$$\Gamma \vdash_H A$$

holds, where Γ is a set of formulas, called **assumptions**.

The Hilbert System Axions:

$$Ax_1 : (\varphi \to (\psi \to \varphi))$$
$$Ax_2 : (\varphi \to (\psi \to \gamma)) \to ((\varphi \to \psi) \to (\varphi \to \gamma))$$
$$Ax_3 : (\neg\varphi \to \neg\psi) \to (\psi \to \varphi)$$
$$MP : \frac{\varphi \quad \varphi \to \psi}{\psi}$$

Note.

Example 2.5. Prove that $\vdash_H (A \to A)$ holds. Note that these steps are not necessarily ordered. In fact, we should not call them steps.

1. $(A \to ((A \to A) \to A))$ (by Ax_1)
   (In axiom 1, replace ψ with $(A \to A)$), replace φ with A)
2. $(A \to ((A \to A) \to A)) \to (A \to (A \to A)) \to (A \to A))$ (by Ax_2)
   (In axiom 2, replace both φ by A and γ with A, ψ by $A \to A$)
3. $(A \to (A \to A)) \to (A \to A))$ (by $MP, 1, 2$)
4. $(A \to (A \to A))$ (by Ax_1)
   (In axiom 1, replace ψ with A)
5. $(A \to A)$ (by $MP, 3, 4$)

Notice that we proved $(A \to A)$ with an empty set of assumptions. This means $(A \to A)$ is a tautology.

Example 2.6. Show that $\{A \to B, B \to C\} \vdash_H (A \to C)$ holds.

1. $(B \to C)$ (by Assumption)
2. $((B \to C) \to (A \to (B \to C)))$ (by Ax_1)
3. $(A \to (B \to C))$ (by $MP, 1, 2$)
4. $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$ (by Ax_2)
5. $((A \to B) \to (A \to C))$ (by $MP, 3, 4$)
6. $(A \to B)$ (by Assumption)
7. $(A \to C)$ (by $MP, 5, 6$)

Definition 2.18 (Deduction Theorem)**.**

$$\Gamma \vdash A \to B \iff \Gamma \cup \{A\} \vdash B$$

Example 2.7. Prove that $\vdash_H (\neg A \to (A \to B))$ holds. You may use the result from Example 2.

1. $(\neg A \to (\neg B \to \neg A))$ (by Ax_1)

2. $(\neg B \to \neg A) \to (A \to B)$ (by Ax_3)
3. $(\neg A \to (A \to B))$ (by $Ex.2, 1, 2$)

Example 2.8. Prove that if $\Sigma \vdash_H A$ and $\Sigma \vdash_H (\neg A)$, then $\Sigma \vdash_H B$ for any B.

1. $(\neg A)$ (by Assumption)
2. $((\neg A) \to ((\neg B) \to (\neg A)))$ (by Ax_1)
3. $((\neg B) \to (\neg A))$ $(MP, 1, 2)$
4. $(((\neg B) \to (\neg A)) \to (A \to B))$ (by Ax_3)
5. $(A \to B)$ (by $MP, 3, 4$)
6. A (by Assumption)
7. B (by $MP, 5, 6$)

Example 2.9. Prove that $\vdash_H (\neg\neg A \to A)$ by applying the Deduction Theorem, we show that $\{(\neg\neg A)\} \vdash_H (A))$

1. $(\neg\neg A)$ (by Assumption)
2. $(\neg\neg A) \to ((\neg\neg\neg\neg A) \to (\neg\neg A))$ (by Ax_1)
3. $((\neg\neg\neg\neg A) \to (\neg\neg A)$ $(MP, 1, 2)$
4. $((\neg\neg\neg\neg A) \to (\neg\neg A)) \to ((\neg A) \to (\neg\neg\neg A))$ (by Ax_3)
5. $(\neg A) \to (\neg\neg\neg A)$ (by $MP, 3, 4$)
6. $((\neg A) \to (\neg\neg\neg A)) \to ((\neg\neg A) \to A)$ (by Ax_3)
7. $(\neg\neg A) \to A$ (by $MP, 5, 6$)
8. A (by $MP, 7, 1$)

Example 2.10. Prove that $\vdash_H (A \to B) \to (\neg B \to \neg A)$ by applying the Deduction Theorem, we show that $\{(A \to B)\} \vdash_H (\neg B \to \neg A))$

1. $(A \to B)$ (by Assumption)
2. $(\neg\neg A) \to A$ (by Ex. 6)
3. $(\neg\neg A) \to B$ (by Ex.2, 1, 2)
4. $B \to (\neg\neg B)$ (proof?)
5. $(\neg\neg A) \to (\neg\neg B)$ (by Ex.2, 3, 4)
6. $((\neg\neg A) \to (\neg\neg B)) \to ((\neg B) \to (\neg A))$ (Ax_3)
7. $((\neg B) \to (\neg A))$ (by $MP, 5, 6$)

Natural Deduction

In general, using proof rules, one can infer a **conclusion** from a set of **premises**.

Let $\Sigma = \{\varphi_1, \varphi_2, \ldots\}$ (for convenience, writen as a sequence $\varphi_1, \varphi_2, \ldots$). Accordingly, the sets $\Sigma \cup \{\varphi\}$ and $\Sigma \cup \Sigma'$ may be written as Σ, φ and Σ, Σ', respectively.

Notation 2.1. We use the symbol $\vdash$ to denote **deducibility** and write

$$\Sigma \vdash \varphi$$

to mean that φ is **deducible** (or **provable**) from Σ.

Natural deduction will be defined by a set of proof **rules**, where **conclusion** φ is derived from a set of **premises** Σ.

Note that $\Sigma \vdash \varphi$ is not a formula (but it can be viewed as a proposition).

Example 2.11. If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. **Therefore**, there were taxis in the station.

This conclusion can be written as follows:

$$(p \wedge \neg q) \to r, \neg r, p \vdash q$$

Constructing such a proof is a creative exercise, a bit like programming.

Roughly speaking, proof rules should have two features:

- One cannot prove invalid patterns of argumentation (called **soundness**)
- Valid arguments can be proved (Called **completeness**)

Example 2.12. We should not be able to show $p, q \vdash p \wedge \neg q$.

The basic rules of natural / formal deduction are shown in the following tables and examples:

Definition 2.19 (natural deduction).

Basic Rules

Name	$\vdash$ Notation	Inference Notation
Reflexivity (Ref)	$\varphi \vdash \varphi$	$\dfrac{\varphi}{\varphi}$
Addition of Premises (+)	If $\Sigma \vdash \varphi$ then $\Sigma \cup \Sigma' \vdash \varphi$	$\dfrac{\frac{\psi}{\varphi}}{\frac{\psi\ \psi'}{\varphi}}$

Example 2.13. Show that if $\varphi \in \Sigma$, then $\Sigma \vdash \varphi$.

Let $\Sigma' = \Sigma - \{\varphi\}$.

(1) $\varphi \vdash \varphi$ (Ref)

(2) $\varphi, \Sigma' \vdash \varphi$ ((+), (1))

(3) $\Sigma \vdash \varphi$

We will call this rule (ϵ).

Rule of Negation

Name	$\vdash$ Notation	Inference Notation
$\neg$-elimination ($\neg-$)	If $\Sigma, \neg\varphi \vdash \psi, \Sigma, \neg\varphi \vdash \neg\psi$ then $\Sigma \vdash \varphi$	$\dfrac{\frac{\neg\varphi}{\psi} \quad \frac{\neg\varphi}{\neg\psi}}{\varphi}$

$\neg-$ means, if we have a contradiction that follows from certain premises (denoted by Σ) with an additional supposition that a certain proposition does not hold (denoted by $\neg\phi$), then this proposition is deducible from the premises (denoted by $\Sigma \vdash \varphi$).

Example 2.14. Show that $\neg\neg\varphi \vdash \varphi$.

$$\dfrac{\dfrac{\neg\neg\varphi \quad \neg\varphi}{\neg\varphi}(\epsilon) \quad \dfrac{\neg\neg\varphi \quad \neg\varphi}{\neg\neg\varphi}(\epsilon)}{\varphi}(\neg-)$$

Another way:

(1) $\neg\neg\varphi, \neg\varphi \vdash \neg\varphi$ (ϵ)

(2) $\neg\neg\varphi, \neg\varphi \vdash \neg\neg\varphi$ (ϵ)

(3) $\neg\neg\varphi \vdash \varphi$ ($\neg-$), (1), (2)

Rule of Conjunction

Name	$\vdash$ Notation	Inference Notation
$\wedge$-introduction ($\wedge+$)	If $\Sigma \vdash \varphi, \Sigma \vdash \psi$, then $\Sigma \vdash \varphi \wedge \psi$	$\dfrac{\varphi \ \psi}{\varphi \wedge \psi}$
$\wedge$-elimination ($\wedge-$)	If $\Sigma \vdash \varphi \wedge \psi$, then $\Sigma \vdash \varphi, \Sigma \vdash \psi$	$\frac{\varphi\wedge\psi}{\varphi}$ $\frac{\varphi\wedge\psi}{\psi}$

$\wedge+$ means, if we have a proof for φ and a proof for ψ, then we have a proof for $\varphi \wedge \psi$. $\wedge-$ says, if we have a proof for $\varphi \wedge \psi$, then we have a proof for φ and a proof for ψ.

Example 2.15. Show that $p \wedge q \vdash q \wedge p$.

$$\dfrac{\frac{p\wedge q}{q}(\wedge-) \quad \frac{p\wedge q}{p}(\wedge-)}{q \wedge p}(\wedge+)$$

Another way:

(1) $p \wedge q \vdash q$ $(\wedge-)$

(2) $p \wedge q \vdash p$ $(\wedge-)$

(3) $p \wedge q \vdash q \wedge p$ $(\wedge+)$, (1), (2)

Example 2.16. Show that $p \wedge q, r \vdash q \wedge r$.

$$\dfrac{\frac{p\wedge q}{q}(\wedge-) \quad r}{q \wedge r}(\wedge+)$$

More specifically:

1. We derive q from $q \wedge r$ by $\wedge$-elimination.
2. Then, we get $q \wedge r$ by $\wedge$-introduction.

Rules of Implication

Name	$\vdash$ Notation	Inference Notation
$\to$-elimination $(\to -)$	If $\Sigma \vdash \varphi \to \psi, \Sigma \vdash \varphi$, then $\Sigma \vdash \psi$	$\dfrac{\varphi \to \psi \quad \varphi}{\psi}$
$\to$-introduction $(\to +)$	If $\Sigma, \varphi \vdash \psi$, then $\Sigma \vdash \varphi \to \psi$	$\dfrac{\frac{\varphi}{\psi}}{\varphi \to \psi}$

Example 2.17. Show that $p, p \to q, p \to (q \to r) \vdash r$.

$$\dfrac{\frac{p\to(q\to r) \quad p}{q\to r}(\to -) \quad \frac{p\to q \quad p}{q}(\to -)}{r}(\to -)$$

1. $p \to (q \to r)$ (premise)
2. $p \to q$ (premise)
3. p (premise)
4. $q \to r$ $(\to -), (1), (3)$
5. q $(\to -), (2), (3)$
6. r $(\to -), (4), (5)$

Example 2.18. Show that $\varphi \to \psi, \psi \to \xi \vdash \varphi \to \xi$.

1. $\varphi \to \psi, \psi \to \xi, \varphi \vdash \varphi \to \psi$ (ϵ)

2. $\varphi \to \psi, \psi \to \xi, \varphi \vdash \varphi \quad (\epsilon)$
3. $\varphi \to \psi, \psi \to \xi, \varphi \vdash \psi \quad (\to -)$, (1),(2)
4. $\varphi \to \psi, \psi \to \xi, \varphi \vdash \psi \to \xi \quad (\epsilon)$
5. $\varphi \to \psi, \psi \to \xi, \varphi \vdash \xi \quad (\to -), (3), (4)$
6. $\varphi \to \psi, \psi \to \xi \vdash \varphi \to \xi \quad (\to +), (5)$

Rules of Disjunction

Name	$\vdash$ Notation	Inference Notation
$\vee$-elimination ($\vee-$)	If $\Sigma, \varphi_1 \vdash \psi, \Sigma, \varphi_2 \vdash \psi,$, then $\Sigma, \varphi_1 \vee \varphi_2 \vdash \psi$	$\dfrac{\frac{\varphi_1}{\psi} \quad \frac{\varphi_2}{\psi}}{\frac{\varphi_1 \vee \varphi_2}{\psi}}$
$\vee$-introduction ($\vee+$)	If $\Sigma \vdash \varphi$, then $\Sigma \vdash \varphi \vee \psi, \Sigma \vdash \psi \vee \varphi$	$\frac{\varphi}{\varphi \vee \psi}$ $\frac{\varphi}{\psi \vee \varphi}$

Check the slides for some examples.

D-Implication

Name	$\vdash$ Notation	Inference Notation
$\leftrightarrow$-elimination ($\leftrightarrow -$)	If $\Sigma \vdash \varphi \leftrightarrow \psi, \Sigma \vdash \varphi$, then $\Sigma \vdash \psi$	$\dfrac{\varphi \leftrightarrow \psi \quad \varphi}{\psi}$
$\leftrightarrow$-introduction ($\leftrightarrow +$)	If $\Sigma, \varphi \vdash \psi, \Sigma, \psi \vdash \varphi$, then $\Sigma \vdash \varphi \leftrightarrow \psi$	$\dfrac{\frac{\varphi}{\psi} \quad \frac{\psi}{\varphi}}{\varphi \leftrightarrow \psi}$

Definition 2.20 (formal proof)**.** Let $\Sigma_1 \vdash \varphi_1, \ldots, \Sigma_n \vdash \varphi_n$ be a sequence, where each $\Sigma_k \vdash \varphi_k$ (for all $1 \leq k \leq n$) is a rule of natural deduction. We say that this sequence is a **formal proof** for $\Sigma_n \vdash \varphi_n$ and φ_n is **formally deducible** from Σ_n.

Note that $\Sigma \models \varphi$ (tautological consequence) and $\Sigma \vdash \varphi$ (deducible) are different matters. The former belongs to semantics while the latter belongs to syntax.

Theorem 2.9. If $\Sigma \vdash \varphi$, then there is some finite $\Sigma^0 \subseteq \Sigma$, such that $\Sigma^0 \vdash \varphi$.

Proof. By structural induction.

□

Theorem 2.10 (transitivity of deducibility)**.** If $\Sigma \vdash \Sigma'$ and $\Sigma' \vdash \varphi$, then $\Sigma \vdash \varphi$. (**transitivity of deducibility**).

Theorem 2.11. 1. $\varphi \to \psi, \varphi \vdash \psi$

2. $\psi \vdash \psi \to \varphi$
3. $\varphi \to \psi, \psi \to \xi \vdash \varphi \to \xi$
4. $\varphi \to (\psi \to \xi), \varphi \to \psi \vdash \varphi \to \xi$

(See Theorems. 2.6.5-2.6.12 in the text book)

2.3 Soundness and Completeness of Natural Deduction

Recall that proof rules of natural deduction are syntactical (i.e., the rules do not know anything about the semantics of formula). We now want to make a connection between proof rules of natural deduction and semantics of propositional logic.

Definition 2.21 (soundness)**.** Soundness of natural deduction means that what we prove using proof rules of natural deduction, is indeed provable. Let Σ be a set of formula and φ be a formula. It means the following:

$$\text{If } \Sigma \vdash \varphi, \quad \text{then} \quad \Sigma \models \varphi$$

This means that natural deduction proof rules, preserve the value of formulas (as the term soundness suggests).

We know that $p \wedge q \models p$, because the valuation that makes $p \wedge q$ true (that ism $p^t = q^t = 1$) also makes p (the right hand side) true. In natural deduction, the $\wedge-$ rule stipulates the same thing: $p \wedge q \vdash p$.

Question. What about $p \vee q \not\models p$?

We now present a proof of soundness.

Proof. By structural induction on the **length of the proof**: 'For all deductions $\Sigma \vdash \psi$ which have a proof of length k, it is the case that $\Sigma \models \psi$' by course-of-values induction on the natural number k.
Base case. The base case of the induction given by the smallest proofs (length 1); they are of the form

$$\Sigma, \varphi \vdash \varphi$$

We need to show that:

$$\Sigma, \varphi \models \varphi$$

This is trivial: recall that any formula (and in particular, φ) is a tautological consequence of a set of formula that includes it (in particular, $\Sigma \cup \{\varphi\}$).
Inductive step. Let us assume that the proof of $\Sigma \vdash \psi$ has k steps and what we want to prove is true for all numbers less than k. Our proof should have the following structure

$$\begin{array}{lll} 1 & \varphi_1 & \text{(premise)} \\ 2 & \varphi_2 & \text{(premise)} \\ & \vdots & \\ n & \varphi_n & \text{(premise)} \\ & \vdots & \\ k & \psi & \text{(justification)} \end{array}$$

There are two things we don't know: (1) what's happening between those dots, and (2) what is the last rule applied. The first is of no concern due to the power of mathematical induction. For the second, we have to consider all rules.

Let $\Sigma = \{\varphi_1, \varphi_2, \dots, \varphi_n\}$:

Let us assume that the last rule is $\wedge+$. Thus, the last step is of the form $\Sigma \vdash \psi_1 \wedge \psi_2$, where ψ_1 and ψ_2 are obtaiend in steps k_1 and k_2, where $k_1, k_2 < k$. Thus, there exists sound proofs for them. That is, $\Sigma \vdash \psi_1$ and $\Sigma \vdash \psi_2$. By the induction hypothesis, we have $\Sigma \models \psi_1$ and $\Sigma \models \psi_2$. This implies $\Sigma \models \psi_1 \wedge \psi_2$.

Let us assume that the last rule is $\rightarrow +$. Thus, the last step is of the form $\Sigma \vdash \psi_1 \rightarrow \psi_2$. Hence, in some step $k' < k$, we must have had $\Sigma, \psi_1 \vdash \psi_2$, for which there exists a proof. By the induction hypothesis, we have $\Sigma, \psi_1 \models \psi_2$. This implies $\Sigma \models \psi_1 \rightarrow \psi_2$. The soundness of the rest of natural deduction rules can be proved in a similar way. □

Definition 2.22 (completeness)**.** Completeness of ND means that if something is provable, then we can prove it using proof rules of natural deduction. Let Σ be a set of formulas and φ be a formula. Formally, **completeness** means the following:

$$\text{If} \quad \Sigma \models \varphi, \quad \text{then} \quad \Sigma \vdash \varphi$$

This means that natural deduction proof rules can prove anything provable by truth tables.

Proof. We prove completeness by showing the contrapositive:

$$\text{If} \quad \Sigma \nvdash \varphi, \quad \text{then} \quad \Sigma \not\models \varphi$$

1. $\Sigma \nvdash \varphi$ implies $\Sigma \cup \{\neg\varphi\}$ is **consistent**
2. implies $\Sigma \cup \{\neg\varphi\}$ has a **model**
3. implies $\Sigma \models \varphi$.

$\Sigma \subseteq (\mathcal{L}^p)$ is **consistent** if and only if there is no $\varphi \in (\mathcal{L}^p)$ such that $\Sigma \vdash \varphi$ and $\Sigma \vdash \neg\varphi$. Consistency is a syntactical notion. As well, $\Sigma \subseteq Form(\mathcal{L}^p)$ is **maximal consistent** if and only if

1. Σ is consistent
2. for any $\varphi \in (\mathcal{L}^p)$ such that $\varphi \notin \Sigma, \Sigma \cup \{\varphi\}$ is inconsistent.

Lemma 2.4. Suppose Σ is a maximal consistent. Then $\varphi \in \Sigma$ if and only if $\Sigma \vdash \varphi$.

Lemma 2.5. If Σ is maximal consistent, then

1. $\neg\varphi \in \Sigma \iff \varphi \notin \Sigma$
2. $\varphi \wedge \psi \in \Sigma \iff \varphi \in \Sigma$ and $\psi \in \Sigma$
3. $\varphi \vee \psi \in \Sigma \iff \varphi \in \Sigma$ or $\psi \in \Sigma$
4. $\varphi \rightarrow \psi \in \Sigma \iff \varphi \in \Sigma$ implies $\psi \in \Sigma$
5. $\varphi \leftrightarrow \psi \in \Sigma \iff \varphi \in \Sigma \iff \psi \in \Sigma$

Lemma 2.6. Suppose Σ is maximal consistent. Then, $\Sigma \vdash \neg\varphi \iff \Sigma \nvdash \varphi$.

Lemma 2.7 (Lindenbaum Lemma)**.** Any consistent set of formulas can be extended to some maximal consistent set.

Theorem 2.12. Suppose $\Sigma \subseteq (\mathcal{L}^p)$. If Σ is consistent, then Σ is satisfiable.

Theorem 2.13 (completeness)**.** Suppose $\Sigma \subseteq (\mathcal{L}^p)$ and $\varphi \in \mathcal{L}^p$. Then

1. If $\Sigma \models \varphi$, then $\Sigma \vdash \varphi$.
2. If $\models \varphi$, then $\vdash \varphi$.

Theorem 2.14. $\Sigma \subseteq (\mathcal{L}^p)$ is satisfiable if and only if each finite subset of Σ is satisfiable.

$\square$

3 Predicate Logic

Definition 3.1 (ordered pair)**.** The **ordered pair** of objects α and β is written as $\langle \alpha, \beta \rangle = \langle \alpha_1, \beta_1 \rangle$ if and only if $\alpha = \alpha_1$ and $\beta = \beta_1$. Similarly, one can define an ordered n-tuple $\langle \alpha_1, \dots, \alpha_n \rangle$. One can also define a set of ordered pairs (for example $\{\langle m, n \rangle \mid m, n$ are natural numbers and $m < N\}$).

Definition 3.2 (cartesian product)**.** The **cartesian product** of sets $S_1, \dots, S_n$ is defined by

$$S_1 \times \cdots \times S_n = \{\langle x_1, \dots, x_n \rangle \mid x_1 \in S_1, \dots, x_n \in S_n\}$$

Let $S^n = \underbrace{S \times \cdots \times S}_{n}$, an n-ary **relation** R on set S is a subset of S^n.

Example 3.1. A special binary relation is the equality relation:

$$\{\langle x, y \rangle \mid x, y \in S \text{ and } x = y\} \equiv \{\langle x, x \rangle \mid x \in S\}$$

For a binary relation R, we often write xRy to denote $\langle x, y \rangle \in R$.

Definition 3.3 (reflexive)**.** R is **reflexive** on S if and only if for any $x \in S, xRx$.

Definition 3.4 (symmetric)**.** R is **symmetric** on S if and only if for any $x, y \in S$, whenever xRy, then yRx.

Definition 3.5 (transitive)**.** R is **transitive** on S if and only if for any $x, y \in S$, whenever xRy and yRz, then xRz.

Definition 3.6 (equivalence relation)**.** R is an **equivalence relation** if and only if R is reflexive, symmetric, and transitive.

Definition 3.7 (R-equivalence)**.** Suppose that R is an equivalence relation on S. For any $x \in S$, the set

$$\bar{x} = \{y \in S \mid xRy\}$$

is called the ***R*-equivalence class of** x. R-equivalence classes make a **partition** of S.

Definition 3.8 (function)**.** A **function (mapping)** f is a set of ordered pairs such that if $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$, then $y = z$.

Definition 3.9 (domain)**.** $dom(f)$ of f is the set $\{x \mid \langle x, y \rangle \in f$ for some $y\}$

Definition 3.10 (range)**.** The **range** $ran(f)$ of f is the set $\{y \mid \langle x, y \rangle \in f$ for some $x\}$

$f(x)$ denotes the unique element in $y \in ran(f)$, where $x \in dom(f)$ and $\langle x, y \rangle \in f$. If f is a function with $dom(f) = S$ and $ran(f) \subseteq T$, we say that f is a function from S to T and denote it by

$$f : S \longrightarrow T$$

Similarly, one can define n-ary functions.

Definition 3.11 (restriction)**.** The **restriction** of R to S_1 is the n-ary relation $R \cap S_1^n$. Suppose $f : S \longrightarrow T$ is a function and $S_1 \subseteq S$. The **restriction** of f to S_1 is the function

$$f \mid S_1 : S_1 \longrightarrow T$$

Definition 3.12 (onto)**.** A function $f : S \longrightarrow T$ is **onto** if $ran(f) = T$.

Definition 3.13 (one-to-one)**.** A function is **one-to-one** if $f(x) = f(y)$ implies $x = y$.

Definition 3.14 (equipotent)**.** Two sets S and T are **equipotent** (that is, $S \sim T$) iff there is a one-to-one mapping from S onto T.

$\sim$ is an equivalence relation.

Definition 3.15 (cardinal)**.** A **cardinal** of a set S is dentoed by $|S|$ where:

$$|S| = |T| \iff S \sim T.$$

Definition 3.16 (countably infinite)**.** A set S is to be **countably infinite**, if and only if $|S| = |\mathbb{N}|$.

Definition 3.17 (countable)**.** A set S is said to be **countable** if and only if $|S| \leq |\mathbb{N}|$ (that is, S is finite or countably infinite).

Theorem 3.1. A subset of a countable set is countable.

Theorem 3.2. The union of any finite number of countable sets if countable.

Theorem 3.3. The union of any countably many countable sets is countable.

Theorem 3.4. The cartesian product of any finite number of countable sets is countable.

Theorem 3.5. The set of all finite sequences with the members of a countable set as components is countable.

3.1 First-Order Predicate Logic

In propositional logic, only the logical forms of compound propositions are analyzed. Propositional logic worked well with statements like not, and, or, if ... then. We need some way to talk about **individuals** (also called **objects**) and refer to some, all, among, and only objects. Propositional logic fails to express such statements. Consider this statement:

Every student is younger than some instructor.

This statement is about being a student, being an instructor, and being younger. These are all properties of some sort that we would like to be able to express along with logical connectives and dependencies. Some more examples:

- For any natural number n, there is a prime number greater than n.
- 2^{100} is a natural number.
- There is a prime number greater than 2^{100}.

Definition 3.18 (first-order logic)**.** **First-order logic** (also called **predicate logic** gives us means to express and reason about objects. It is a scientific theory with these ingredients:

- Domain of objects (individuals) (e.g., the set of natural numbers)

- Variables
- Designated individuals (e.g., '0')
- Functions (e.g., '+' and '.')
- Relations (e.g., '=')
- Quantifiers and Propositionsl connectives

Now, let's explore the meaning of each of these.
We use **predicates** (that is, relations) to express statements such as 'being a student'. For example, we could write $S(liam)$ to denote that Liam is a student and $I(borzoo)$ to denote that Borzoo is an instructor. Likewise, $Y(liam, borzoo)$ could mean that Liam is younger than Borzoo. In order to make predicates more expressive, we use **variables**. Think of variables as **place holders** that can be replaced by concrete objects.
For example:

- $S(x)$: x is a student
- $I(x)$: x is an instructor
- $Y(x, y)$: x is younger than y

Notice that we cab write the meaning of I or S by using any variable instead of x, such as y or z. In general, we use variables that range over a **domain of objecs** to make general statements

$$x^2 \geq 0$$

and in expressing conditions which individuals may or may not satisfy:

$$x + x = x \cdot x$$

This condition is satified by only 0 and 2.
We need to convey the meaning of '**Every** student x is younger than **some** professor y'. This is where we use the terms **for all** and **there exists** frequently (called **quantifiers**). For example:

- For all $\epsilon > 0$, there exists some $\delta > 0$ such that if $|x - a| < \delta$, then $|f(x) - b| < \epsilon$.
- "For all" is called the **universal quanitifer** $\forall$, and
- "There exists" is the **existential quantifer** $\exists$.

A quantiier is always attached to variables as in $\forall x$ (for all x) and $\exists x$ (there exists z). We can now write our examples entirely symbolically (although paraphrased!):

$$\forall x.(S(x) \rightarrow (\exists y.(I(y) \wedge Y(x, y))))$$

Or, the statement 'Not all birds can fly' can be written as:

$$\neg(\forall x(B(x) \rightarrow F(x)))$$

In addition to predicates and quantifiers, first-order logic extends propositional logic by using **functions** as well. Consider the following statement:

Every child if younger than its mother.

One way to express this statement in first-order logic is the following:

$$\forall x.\forall y(C(x) \land M(y,x) \rightarrow Y(x,y))$$

But this means x can have multiple mothers!
Functions in first-order logic gives us way to express statements more concisely. The previous example can be expressed as follows:

$$\forall x(C(x) \rightarrow Y(x, m(x)))$$

where m is a function: it takes one argument and returns the mother of that argument.

More examples:

- Andy and Paul have the same maternal grandmother

$$m(m(a)) = m(m(p))$$

- Ann likes Mary's brother:

$$\exists x(B(x,m) \land L(a,x))$$

Consider:

For all x, x is even.
There exists x such that x is even.

Since x ranges over $\mathbb{N}$, they mean:

For all natural numbers x, x is even.
There exists a natural number x such that x is even.

These have truth values! Also,

'4 is even'

is a proposition since 4 is an individual in $\mathbb{N}$. If we replace 4 by a variable x ranging over $\mathbb{N}$, then

'x is even'

is not a proposition and has no truth value. It is a proposition function.

Definition 3.19 (proposition function)**.** A **proposition function** on a domain D is an n-ary function mapping D^n into $\{0,1\}$.

3.2 Syntax of Predicate Logic

- Constant (individual) symbols (CS): $c, d, c_1, c_2, \ldots, d_1, d_2, \ldots$
- Function Symbols (FS): $f, g, h, f_1, f_2, \ldots, g_1, g_2$
- Variables (VS): $x, y, z, x_1, x_2, \ldots, y_1, y_2, \ldots$
- Predicate (Relational) Symbols (PS): $P, Q, P_1, P_2, \ldots, Q_1, Q_2, \ldots$

- Logical Connectives $\neg, \wedge, \vee, \implies$
- Quantifiers $\forall$ (for all) and $\exists$ (there exists)
- Punctuation: '(',')', '.', and ','.

Example 3.2.

0: constant '0'

S: function (successor) $S(x)$ stands for '$x+1$'

Eq: relation (equality) $Eq(x,y)$ stands for: '$x=y$'

plus: function (addition) $plus(x,y)$ stands for: '$x+y$'

$$\forall x.Eq(plus(x,S(S(0))),S(S(x)))$$

means "Adding two to a number results in the second successor of that number"

Example 3.3.

$$\forall x.\forall y.Eq(plus(x,y),plus(y,x))$$

means "Addition is commutative".

$$\neg\exists x.Eq(0,S(x))$$

means "0 is not the successor of any number."

Definition 3.20 (term)**.** The set $Terms(\mathcal{L})$ of **terms** of $\mathcal{L}$ is defined using the following rules:

- All constants in CS are terms
- All variables in VS are terms
- If $t_1,\ldots,t_n \in Term(\mathcal{L})$ and f is an n-ary function, then $f(t_1,\ldots,t_n) \in Term(\mathcal{L})$.

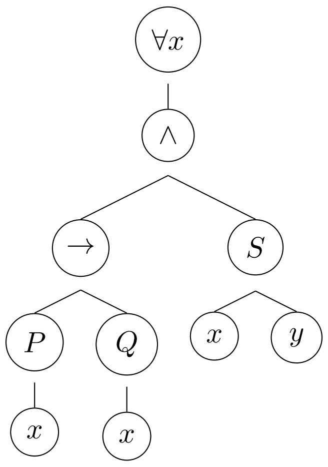
Definition 3.21 (atom)**.** Let P be a predicate (that is, an n-ary relation). An expression of $\mathcal{L}$ is an **atom** in $Atoms(\mathcal{L})$ if and only if it is of one of the forms $P(t_1,\ldots,t_n)$ where $t_1,\ldots,t_n$ are terms in $Term(\mathcal{L})$.

Definition 3.22 (formula)**.** We define the set $Form(\mathcal{L})$ of first-order logic formulas inductively as follows:

1. $Atom(\mathcal{L}) \subseteq Form(\mathcal{L})$
2. If $\varphi \in Form(\mathcal{L})$, then $(\neg\varphi) \in Form(\mathcal{L})$
3. If $\varphi,\psi \in Form(\mathcal{L})$, then $(\varphi * \psi) \in Form(\mathcal{L})$, where $* \in \{\wedge,\vee,\implies\}$
4. If $\varphi \in Form(\mathcal{L})$ and $x \in VS$, then $\forall x.\varphi \in Form(\mathcal{L})$ and $(\exists x.\varphi) \in Form(\mathcal{L})$

Parse trees are similar to propositional formula:

- Quantifiers $\forall x$ and $\exists y$ form nodes like negation (i.e., only one sub-tree)
- Predicates $P(t_1,t_2,\ldots,t_n)$ has P as a node and terms $t_1,t_2,\ldots,t_n$ as children nodes.



$$\forall x.((P(x) \to Q(x)) \wedge S(x, y))$$

How is the following formula generated?

$$\forall x.(F(b) \implies \exists y.(\forall z.G(y, z) \vee H(u, x, y)))$$

To evaluate first-order formulas, we need to understand the nature of occurrence of variables. For example, in this above parse tree,

- three leaves labeled by x: if we walk up from these nodes, we reach a node labeled by $\forall x$
- one leaf labeled by y: if we walk up from this node, we will reach no quantifiers for y

Definition 3.23 (free)**.** We say that an occurrence of x is **free** in first-order formula φ, if in the parse tree of φ, there is no upwards path from x to a node labeled by $\forall x$ or $\exists x$.

Definition 3.24 (quantified)**.** An occurrence of x that is not free is called **bound** or **quantified**.

Definition 3.25 (free variable)**.** Let $\varphi \in Form(\mathcal{L})$. We define the set $FV(A)$ of **free variables** of A as follows:

1. $\{x \mid x \text{ appears in } t_i \text{ for some } 0 < i \leq ar(P)\}$, for $\varphi = P(t_1, \ldots, t_{ar(P)})$
2. $FV(\varphi)$ for $\psi = (\neg\varphi)$
3. $FV(\varphi) \cup FV(\psi)$ for $\gamma = (\varphi * \psi)$, where $* \in \{\wedge, \vee \implies\}$
4. $FV(\varphi) - \{x\}$ for $\psi = (\forall x.\varphi)$ or $\psi = (\exists x.\varphi)$

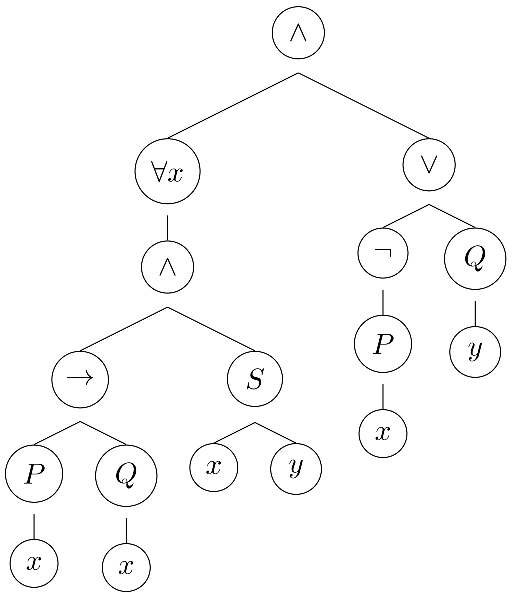
Variables not in $FV(\varphi)$ are **bound variables**.

Definition 3.26 (scope)**.** If $\forall x.A(x)$ or $\exists x.A(x)$ is a segment of B, $A(x)$ is called the **scope** in B of the $\forall x$ or $\exists x$ on the left of $A(x)$.

In the following formula:

$$\exists x.\forall y.\exists z.F(x, y, z)$$

what is the scope of $\forall y$?



$$(\forall x.((P(x) \wedge Q(x))) \to (\neg P(x) \vee Q(y))$$

Is x free or quantified?

Definition 3.27 (closed)**.** A formula $A \in Form(\mathcal{L})$ is **closed** (also called a **sentence**) if $FV(A) = \{\}$.

Definition 3.28 (substitution)**.** Given a variable x, a term t, and a formula φ, we define $\varphi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in φ with t.

Example 3.4. Consider formula

$$\varphi = \forall x.((P(x) \to Q(x)) \wedge S(x, y))$$

We have

$$\varphi[f(x, y)/x] = \varphi$$

because there is no free occurrence of x.

Example 3.5. Consider formula

$$\varphi = (\forall x.((P(x) \wedge Q(x)))) \to (\neg P(x) \vee Q(y))$$

We have

$$\varphi[f(x, y)/x] = (\forall x.((P(x) \wedge Q(x)))) \to (\neg P(f(x, y)) \vee Q(y))$$

We say that the term t is **free for** variable x in formula φ is in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

Example 3.6. Consider term $t = f(y, y)$ and formula

$$\varphi = S(x) \wedge (\forall y.(P(x) \to Q(y)))$$

The leftmost x can be substituted by t since it is not in the scope of any quantifier, but substituting the rightmost x introduces a new variable y in t, which becomes bound by $\forall y$. Hence, $f(y, y)$ is not free for x in φ. Such cases can be resolved by variable renaming, for example $t = f(z, z)$.

1. For a term t_1, $(t_1)[t/x]$ is t_1 with each occurrence of the variable x replaced by the term t.
2. For $\varphi = P(t_1, \ldots, t_{ar(P)})$, $(\varphi)[t/x] = P((t_1)[t/x], \ldots, (t_{ar(P)})[t/x])$.
3. For $\varphi = (\neg\psi)$, $(\varphi)[t/x] = (\neg(\psi)[t/x])$;
4. For $\varphi = (\psi \to \eta)$, $(\varphi)[t/x] = ((\psi)[t/x] \to (\eta)[t/x])$, and
5. For $\varphi = (\forall y.\psi)$, there are two cases
   - if x is y, then $(\varphi)[t/x] = \varphi = (\forall y.\psi)$ and
   - otherwise, then $(\varphi)[t/x] = (\forall z.(\psi[z/y])(t/x)$, where z is any variable that is not free in t or in φ.

In the last case above, the additional substitution $(.)[z/y]$ (that is, renaming the variable y to z in ψ) is needed in order to avoid an accidental **capture a variable** by the quantifier (that is, captrue of any y that is possibly free in t).

3.3 Semantics of Predicate Logic

In propositional logic, semantics was described in terms of valuation of (the only ingredients) propositional variables. The first-order language includes more ingrediants (i.e., predicates and functions) and, hence, the interpretations for it are more complicated. First-order formula are intended to express propositions (i.e, true/false valuation). This is accomplished by **interpretations** (also called **models**).

Definition 3.29 (interpretation)**.** A first order **interpretation** I is a tuple $(D, (.)^I)$:

- D is a non-empty set called the **domain** (or **universe**); and
- $(.)^I$ is an **interpretation function** that maps
  - constant symbols $c \in CS$ to individuals $c^I \in D$
  - function symbols $f \in FS$ to functions $f^I : D^{ar(f)} \to D$; and
  - predicate symbols $P \in PS$ to relations $P^I \subseteq D^{ar(P)}$.

Example 3.7. Let functions f and g be respectively addition and squaring functions and P be the equality relation. Let $P(f(g(a), g(b)), g(c))$ be a closed formula, where individuals a,b, and c be interpreted as 4, 5, and 6 in $\mathbb{N}$.Then, the above predicate is interpreted as the false proposition.

Example 3.8. Let $f(g(a), f(b, c))$ be a term. Let individuals a, b, and c be interpreted as 4, 5, and 6 in $\mathbb{N}$ and functions f and g are respectively as addition and squaring. Then, the above term is interpreted as $4^2 + (5 + 6)$ which is the individual 27 in $\mathbb{N}$.

Example 3.9. interpretation is extremely liberal and openended. For example, consider the non-closed formula:

$$P(f(g(u), g(b)), g(w))$$

where only b is interpreted as 5. One can interpret this formula by:

$$x^2 + 5^2 = y^2$$

where x and y are free variables. This is not a proposition, but a binary proposition function in $\mathbb{N}$.

Given an interpretation, in order to evaluate the truthfulness of a formula $\forall x.\varphi$ or $\exists x.\varphi$, we should check whether φ holds for all or some value a in the interpretation. The mechanism to check this is by using substitution $\varphi[a/x]$ for values a in an interpretation. This is called a **valuation**. For example, in the previous example, one can obtain a truth value by assigning individuals in $\mathbb{N}$ to x and y.

Definition 3.30 (valuation)**.** A **valuation** θ (also called an **assignment**) is a mapping from VS, the set of variables, to domain D. For example, the non-closed formula

$$x^2 + 4^2 = y^2$$

$\theta(x) = 3$ and $\theta(y) = 5$ evaluated the formula to the true proposition.

Let I be a first order interpretation and θ a valuation. For a term t in $Term(\mathcal{L})$, we define interpretation and valuation of t, $t^{I,\theta}$, as follows:

1. $c^{I,\theta} = c^I$ for $t \in CS$ (i.e., t is a constant);
2. $x^{I,\theta} = \theta(x)$ for $t \in VS$ (i.e., t is a variable); and
3. $f(t_1, \ldots, t_{ar(f)})^{I,\theta} = f^I((t_1)^{I,\theta}, \ldots, t_{ar(f)}^{I,\theta})$, otherwise (i.e., for t a functional term).

Example 3.10. Suppose a language has a constant symbol 0, a unary function s, and a binary function $+$. Let us write $+$ in infix position (i.e., $x + y$ instead of $+(x, y)$). Notice that $s(s(0) + s(x))$ and $s(x, s(x + s(0)))$ are two terms. The following are examples of interpretations and valuations:

- $D = \{0, 1, 2, \ldots\}, 0^I = 0, s^I$ is the successor function and $+^I$ is the addition operation. Then if $\theta(x) = 3$, $s(s(0) + s(x)) = 6$ and $s(x, s(x + s(0))) = 9$.
- D is the collection of all words over the alphabet $\{a, b\}, 0^I = a, s^I$ is the operation that appends a to the end of the word, and $+^I$ is the concatenation. Then if $\theta(x) = aba$, $s(s(0) + s(x)) = aaabaaa$ and $s(x, s(x + s(0))) = abaabaaaaa$.
- $D = \{\ldots, -2, -1, 0, 1, 2, \ldots\}, 0^I = 1, s^I$ is the predecessor function and $+^I$ is the subtraction operation. Then, in general, $s(s(0) + s(x)) = -\theta(x)$ and $s(x + s(x + s(0))) = 0$, given any valuation θ.

Definition 3.31 (satisfaction relation)**.** The **satisfaction relation** $\models$ between an interpretation I, a valuation θ, and a first-order formula φ is defined as:

- $I, \theta \models P(t_1, \ldots, t_{ar(P)})$ iff $\langle (t_1)^{I,\theta}, \ldots, (t_{ar(P)})^{I,\theta} \rangle \in P^I$ for $P \in PS$
- $I, \theta \models \neg\varphi$ if and only if $I, \theta \models \varphi$ is not true
- $I, \theta \models \varphi \wedge \psi$ if and only if $I, \theta \models \varphi$ and $I, \theta \models \psi$
- $I, \theta \models \varphi \vee \psi$ if and only if $I, \theta \models \varphi$ or $I, \theta \models \psi$

Remark 3.1.

1. $\langle (t_1)^{I,\theta}, \ldots, (t_{ar(P)})^{I,\theta} \rangle \in P^I$ means that $(t_1)^{I,\theta}, \ldots, (t_{ar(P)})^{I,\theta}$ is in the relation P^I
2. If $A(x)$ is a variable with no free occurrence of u and $A(u)$ is a formula with no free occurrence of x, then $A(x)$ and $A(u)$ have the same intuitive meaning.
3. For the same reason, $\forall x.A(x)$ and $\forall u.A(u)$ have the same meaning.

One can trivially define the following:

- $I, \theta \models (\forall x.\varphi)$ if and only if $I, \theta([x = v]) \models \varphi$ for all $v \in D$
- $I, \theta \models (\exists x.\varphi)$ if and only if $I, \theta([x = v]) \models \varphi$ for some $v \in D$

where the valuation $[x = v](y)$ is defined to be v when $x = y$ and θ otherwise.

Example 3.11. Let *loves* be a binary predicate. Consider the following formula:

$$\forall x.\forall y(loves(x, alma) \wedge loves(y, x) \rightarrow \neg loves(y, alma))$$

Let interpretation I be the following: $D = \{a, b, c\}$, $loves^I = \{(a, a), (b, a), (c, a)\}, CS = \{alma\}$, $alma^I = a$. The above formulas intends to capture the expression:

None of Alma's lovers' lovers love her.

This is not the case!

Example 3.12. Suppose R is a binary relation and $\oplus$ is a binary function.

- Consider the sentence $\exists y.R(x, y \oplus y)$. Suppose $D = \{1, 2, 3, \ldots\}$, $\oplus^I$ is the addition operation, and R^I is the equality relation. Then, $I, \theta \models \exists y.R(x, y \oplus y)$ iff $\theta(x)$ is an even number.

The universal and existential quantifiers may be interpreted respectively as a generalization of conjunction and disjunction. If the domain $D = \{\alpha_1, \ldots, \alpha_k\}$ is finite then:

For all x such that $f(x)$ iff $R(\alpha_1)$ and ... and $R(\alpha_k)$
There exists x such that $R(x)$ iff $R(\alpha_1)$ or ... or $R(\alpha_k)$

where R is a property.

Lemma 3.1. Let φ be a first order formula, I be an interpretation, and θ_1 and θ_2 be two valuations such that $\theta_1(x) = \theta_2(x)$ for all $x \in VS$. Then,

$$I, \theta_1 \models \varphi \qquad I, \theta_2 \models \varphi$$

Proof by structural induction.

Definition 3.32 (satisfiable)**.** $\Sigma \subseteq Form(\mathcal{L})$ is **satisfiable** iff there is some interpretation I and valuation θ such that $I, \theta \models \varphi$ for all $\varphi \in \Sigma$.

Definition 3.33 (valid)**.** A formula $\varphi \in Form(\mathcal{L})$ is **valid** iff for all interpretations I and valuation θ, we have $I, \theta \models \varphi$.

Example 3.13. Let $\varphi = P(f(g(x), g(y)), g(z))$ be a formula. The formula is satisfiable:

- f^I = summation
- g^I = squaring
- P^I = equality
- $\theta(x) = 3, \theta(y) = 4, \theta(z) = 5$

φ is not valid.

Definition 3.34 (logical consequence)**.** Suppose $\Sigma \subseteq Form(\mathcal{L})$ and $\varphi \in Form(\mathcal{L})$. We say that φ is a **logical consequence** of Σ (that is, of the formula in Σ), written as $\Sigma \models \varphi$ iff for any interpretation I and valuation θ, we have $I, \theta \models \Sigma$ implies $I, \theta \models \varphi$.

$\models \varphi$ means that φ is valid.

Example 3.14. Show that $\models \forall x.(\varphi \to \psi) \to ((\forall x.\varphi) \to (\forall x.\psi))$

Proof by contradiction. Suppose there exists I and θ such that,

$$\begin{aligned}
&I, \theta \not\models \forall x.(\varphi \to \psi) \to ((\forall x.\varphi) \to (\forall x.\psi)) \\
&I, \theta \models \forall x.(\varphi \to \psi) \\
&I, \theta \models \forall x.\varphi \\
&I, \theta \not\models \forall x.\psi
\end{aligned}$$

$$\begin{aligned}
I, \theta([x = v]) &\models \varphi \\
I, \theta([x = v]) &\not\models \psi \\
I, \theta([x = v]) &\not\models \varphi \to \psi \\
I, \theta &\not\models \forall x.(\varphi \to \psi) \qquad \text{(contradiction)}
\end{aligned}$$

Example 3.15. Show that $\forall x.\neg A(x) \models \neg\exists x.A(x)$

Proof by contradiction. Suppose there exists I and θ such that,

$$\begin{aligned}
&I, \theta \models \forall x.\neg A(x) \quad \text{and} \quad I, \theta \not\models \neg\exists x.A(x) \\
&I, \theta \models \exists x.A(x)
\end{aligned}$$

$$\begin{aligned}
&I, \theta([x = v]) \models \neg A(x) \quad \text{for all } v \\
&I, \theta([x = v]) \models A(x) \quad \text{for some } v
\end{aligned}$$

Contradiction!

$$\begin{aligned}
&q \vdash_{ND} q \\
&\vdash NDp \lor \neg p \\
q \vdash q \land (p \lor \neg p)
\end{aligned}$$

Be familiar with soundness and completeness proofs.

3.4 Proof Systems in First-order Logic

Proof calculi for predicate logic are similar to those for propositional logic, except that we have new proof rules for dealing with the quanitifiers. Again, we explore

- Hilbert Systems, and

- Natural Deduction

First-Order Logic Hilbert System

Ax_1	$\langle \forall^* (\varphi \to (\psi \to \varphi)) \rangle$
Ax_2	$\langle \forall^* ((\varphi \to (\psi \to \eta)) \to ((\varphi \to \psi) \to (\varphi \to \eta))) \rangle$
Ax_3	$\langle \forall^* (((\neg\varphi) \to (\neg\psi)) \to (\psi \to \varphi)) \rangle$
Ax_4	$\langle \forall^* (\forall x.(\varphi \to \psi)) \to ((\forall x.\varphi) \to (\forall x.\psi)) \rangle$
Ax_5	$\langle \forall^* (\forall x.\varphi) \to \varphi[x/t] \rangle$ for $t \in T$ a term
Ax_6	$\langle \forall^* (\varphi \to \forall x.\varphi) \rangle$ for $x \notin FV(\varphi)$
MP	$\langle \varphi, (\varphi \to \psi), \psi \rangle$

where $\forall^*$ is a finite sequence of universal quantifiers (e.g., $\forall x_1.\forall y.\forall x$).

Example 3.16. Show that $\vdash \forall x.\forall y.\varphi \to \forall y.\forall x.\varphi$.

(1)	$\forall x.\forall y.\varphi$	Deduction Theorem
(2)	$\forall x.\forall y.\varphi \to (\forall y.\varphi)[x/t]$	Ax_5
(3)	$(\forall y.\varphi)[x/t]$	MP
(4)	$(\forall y.\varphi)[x/t] \to ((\varphi)[x/t])[y/t']$	Ax_5
(5)	$((\varphi)[x/t])[y/t']$	MP
(6)	$((\varphi)[x/t])[y/t'] \to \forall x.(\varphi)[x/t]$	Ax_6
(7)	$\forall x.(\varphi)[x/t]$	MP
(8)	$\forall x.(\varphi)[x/t] \to \forall y.\forall x.\varphi$	Ax_6
(9)	$\forall y.\forall x.\varphi$	MP

Example 3.17. Show that $\vdash A(a) \to \exists x.A(x)$.

(1)	$\forall x.\neg A(x) \to \neg A(a)$	Ax_5
(2)	$A(a) \to (\neg\forall x.\neg A(x))$	Ax_3
(3)	$A(a) \to \exists x.A(x)$	Definition of $\exists$

We can prove the existential quantifier by taking $\forall x.\phi \models \neg\exists x.\neg\phi$ by assuming $\forall x.\phi$ and $\exists x.\neg\phi$ (the negation of the conclusion), and following through to find a contradiction. This is left as an exercise.

Example 3.18. Show that $\vdash \forall x.(A(x) \to B(x)) \to (\forall x.A(x) \to \forall x.B(x))$.

(1)	$\forall x.(A(x) \to B(x))$	Assumption
(2)	$\forall x.A(x)$	Assumption
(3)	$\forall x.A(x) \to A(a)$	Ax_5

(4)	$A(a)$	$MP, 2, 3$
(5)	$\forall x.(A(x) \to B(x)) \to (A(a) \to B(a))$	Ax_5
(6)	$A(a) \to B(a)$	$MP, 1, 5$
(7)	$B(a)$	$MP, 4, 6$
(8)	$B(a) \to \forall x.B(x)$	Ax_6

Soundness of FOL Hilbert System

Step 1: Satisfiability (satisfiable) and validity (valid).

Suppose $\Sigma \subseteq Form(\mathcal{L}), A \in Form(\mathcal{L})$, and D is a domain.

1. Σ is satisfiable in D iff there is some model I, θ over D such that $I, \theta \models \varphi$ for all $\varphi \in \Sigma$.
2. A is valid in D iff for all models I, θ over D, we have $I, \theta \models A$.

Theorem 3.6. Suppose formula A contains no equality symbol and $|D| \leq |D_1|$.

- If A is satisfiable in D, then A is satisfiable in D_1.
- If A is valid in D_1, then A is valid in D.

Theorem 3.7 (soundness)**.** • If $\Sigma \vdash A$, then $\Sigma \models A$.

- If $\vdash A$, then $\models A$.

That is, every formally provable formula is valid.

Definition 3.35 (consistent)**.** We say that $\Sigma \subseteq Form(\mathcal{L})$ is **consistent** iff there is no $A \in Form(\mathcal{L})$ such that $\Sigma \vdash A$ and $\Sigma \vdash \neg A$. Consistency is a syntactical notion.

Theorem 3.8. If Σ is satisfiable, then Σ is consistent.

Definition 3.36 (maximal consistent)**.** We say that $\Sigma \subseteq Form(\mathcal{L})$ is **maximal consistent** iff

- Σ is consistent
- for any $A \in Form(\mathcal{L})$ such that $A \notin \Sigma$, $\Sigma \cup \{A\}$ is inconsitent.

Lemma 3.2. Suppose Σ is maximal consistent. Then, $A \in \Sigma$ iff $\Sigma \vdash A$.

Lemma 3.3 (Lindenbaum Lemma)**.** Any consistent set of formula can be extended to some maximal consistent set.

Theorem 3.9. Suppose $\Sigma \subseteq Form(\mathcal{L})$. If Σ is consistent, then Σ is satisfiable.

Theorem 3.10. Suppose $\Sigma \subseteq Form(\mathcal{L})$ and $A \in Form(\mathcal{L})$. Then

- if $\Sigma \models A$, then $\Sigma \vdash A$.
- if $\models A$, then $\vdash A$.

3.5 Natural Deduction in First-Order Logic

Natural deduction in first-order logic is similar to propositional logic except we need to introduce rules for quantifier elimination and introduction. Other proof techniques and tricks remain the same as natural deduction for propositional logic.

Name	$\vdash$ Notation	Inference Notation
$\forall$-elimination ($\forall-$)	If $\Sigma \vdash \forall x.\varphi$ then $\Sigma \vdash \phi[x/t]$	$\dfrac{\forall x.\phi}{\varphi[x/t]}$
$\forall$-introduction ($\forall+$)	If $\Sigma \vdash \varphi[x/u]$ then $\Sigma \vdash \forall x.\varphi$	$\dfrac{\varphi[x/u]}{\forall x.\varphi}$

In ($\forall-$), the formula $\varphi[x/t]$ is obtained by substituting t for all occurrences of x. In ($\forall+$), u should not occur in Σ.

The rule ($\forall+$) is a bit tricky. Think of it this way: if you want me to prove that $\forall x.\varphi$, then I show the truthfulness of φ for any 'random' x you give me. In other words, if we prove φ about any u that is not special in any way, then you can prove it for any x whatsoever. That is, the step from φ to $\forall x.\varphi$ is legitimate if only we have arrived at φ in such a way that none of its assumptions contain x as a free variable.

Rules of elimination and introduction in first-order logic natural deduction can be generalized to multiple quantifiers:

- If $\Sigma \vdash \forall x_1 \ldots x_n.\varphi$ then $\Sigma \vdash [x_1/t_1 \ldots x_n/t_n]$.
- If $\Sigma \vdash \varphi[x_1/u_1 \ldots x_n/u_n]$ then $\Sigma \vdash \forall x_1 \ldots x_n.\varphi$, where $u_1, \ldots, u_n$ do not occur in Σ.

Example 3.19. Show that $\forall x.\forall y.A(x,y) \vdash \forall y.\forall x.A(x,y)$

1. $\forall xy.\varphi(x,y) \vdash \varphi(u,v)$
2. $\forall xy.\varphi(x,y) \vdash \forall yx.\varphi(x,y)$ (Generalized $\forall+$)

In Step 1, u and v should not occur in $\varphi(x,y)$.

Name	$\vdash$ Notation	Inference Notation
$\exists$-elimination ($\exists-$)	If $\Sigma, \varphi(u) \vdash \psi$ then $\Sigma, \exists x.\varphi(x) \vdash \psi$	$\dfrac{\dfrac{\varphi(u)}{\psi} \quad \exists x.\varphi(x)}{\psi}$
$\exists$-introduction ($\exists+$)	If $\Sigma \vdash \varphi[x/t]$ then $\Sigma \vdash \exists x.\varphi$	$\dfrac{\varphi[x/t]}{\exists x.\varphi}$

In ($\exists-$), u should not occur in Σ or ψ. In ($\exists+$), $\varphi(x)$ is obtained by replacing some occurences of t in φ by x. In the ($\exists+$) rule notice that $\varphi[x/t]$ has more information that $\exists x.\varphi$. For example, let $t = y$ such that $\varphi[x/t]$ is $y = y$. Then, φ could be a number of things, such as $x = x$ or $x = y$.

Example 3.20. Show that $\exists x.\varphi(x) \vdash \exists y.\varphi(y)$

(1) $\varphi(u) \vdash \varphi(u)$ Ref

(2) $\varphi(u) \vdash \exists y.\varphi(y)$ $(\exists+)$

(3) $\exists x.\varphi(x) \vdash \exists y.\phi(y)$ $(\exists-)$

Example 3.21. Show that $\neg\forall x.\varphi(x) \vdash \exists x.\neg\varphi(x)$.

(1) $\neg\varphi(u) \vdash \exists x.\neg\varphi(x)$ (u not occuring in $\varphi(x)$)

(2) $\neg\exists x.\neg\varphi(x) \vdash \varphi(u)$ (1)

(3) $\neg exists.\neg\varphi(x) \vdash \forall x.\varphi(x)$

(4) $\neg\forall x.\varphi(x) \vdash \exists x.\neg\varphi(x)$

Example 3.22. Show that $\forall x.\varphi(x) \to \psi \vdash \exists x.(\phi(x) \to \psi)$, x not occuring in ψ.

Definition 3.37 (First-Order Axioms of Equality)**.** Let $\approx$ be a binary predicate (written in infix). We define the **First-Order Axioms of Equality** as follows:

- EqId : $\langle \forall x.(x \approx x)\rangle$;
- EqCong : $\langle \forall x.\forall y.(x \approx y) \to (\varphi^z_x \to \varphi^z_y)\rangle$;

4 Programs

Definition 4.1 (equality relation)**.** The **equality relation** $\approx$ over a domain $\mathcal{D}$ is the binary relation $\{\langle x, x\rangle \mid x \in \mathcal{D}\}$.

Example 4.1. $\mathcal{D} = \{a, b, c\}$
Eq: $\mathcal{D} \to \mathcal{D}$ and Eq $= \{\langle a, a\rangle\langle b, b\rangle\langle c, c\rangle\}$.

The equality relation satisfies the following axioms:

EQ1. $\forall x.x \approx x$ "everything is equal to itself".

EQ2. For each $\phi \in Form(\mathcal{L})$ and every variable z,

$$\forall x.\forall y.(x \approx y \to (\varphi[z/x] \to \varphi[z/y]))$$

"equal objects have the same properties".

Lemma 4.1. The axioms for equality imply that the relation $\approx$ is symmetric and transitive.

$$\vdash \forall x.\forall y.(x \approx y \to y \approx x)$$

$$\vdash \forall w.\forall x.\forall y.(x \approx y \to (y \approx w \to x \approx w))$$

We can prove symmetry using natural deduction,

(1) $b \approx b$ EQ1

(2) $a \approx b \to (b \approx b \to b \approx a)$ EQ2

(3) $\boxed{a \approx b}$ Assumption

(4) $b \approx b \to a \approx a$ $(\to -)$, 2, 3

(5) $b \approx a$ $(\to -)$, 1, 3

(6) $a \approx b \to b \approx a$

(7) $\forall x.\forall y.(x \approx y \to y \approx x)$

Additionally we can show transitivity,

(1) $b \approx a \to (b \approx c \to a \approx c)$ EQ2

(2) $\boxed{a \approx b}$ Assmption

(3) $a \approx b \to b \approx a$ Symmetry

(4) $b \approx a$ $(\to -)$

(5) $b \approx c \to a \approx c$ $(\to -)$

(6) $a \approx b \to (b \approx c \to a \approx c)$

(7) $\forall x.\forall y.(x \approx y \to (y \approx a \to x \approx a))$

Natural numbers derive from the fundamental notion of counting or ordering objects, where '0' stands for 'no objects'. Once we have counted some objects, if we have a new objects, we have a new number of objects and this number is different from all the previous numbers. This way we can generate the set of all natural numbers. Consider S the unary function $S : \mathbb{N} \to \mathbb{N}$ where $S(x)$ is the 'next number' (successor) of $x \in \mathbb{N}$. Starting with '0' and using the function S, we have a term for every natural number.

A standard signature of arithmetic is $\sigma = \langle 0, S, +, \cdot, \approx, \not\approx \rangle$. The standard axiomatization of arithemetics is called **Peano arithmetic**.

PA1 $\forall x.S(x) \not\approx 0$

PA2 $\forall x.\forall y.(S(x) \approx S(y) \to x \approx y)$

PA3 $\forall x.(x + 0 \approx x)$

PA4 $\forall x.\forall y.(x + S(y)) \approx S(x + y)$

PA5 $\forall x.(x \cdot 0 \approx 0)$

PA6 $\forall x.\forall y.(x \cdot S(y) \approx s \cdot y + x)$

PA7 For each variable x and every formula $\varphi(x)$,

$$\varphi[x/0] \to ((\forall x.(\varphi \to \varphi[x/S(x)])) \to \forall x.\varphi)$$

Lemma 4.2. Peano Arithmetic has a proof of the formulas

$$\forall y.(0 + y \approx y)$$

Lemma 4.3. For each free variable x

$$\forall y.(x + y \approx y + x) \vdash_{PA} \forall y.(S(x) + y) \approx S(x + y)$$

Theorem 4.1. Addition in Peano Arithmetic is commutative, that is, there exists a proof of

$$\vdash \forall x.\forall y.(x + y \approx y + x)$$

from the axioms PA1 - PA7.

Example 4.2. The statement "every non-zero natural number has a predecessor" can be expressed by the formula

$$\vdash_{PA} \forall x.(x \not\approx 0 \to \exists y.S(y) \approx x)$$

Show that this formula has a proof from the PA axioms.

Exercise. Write the formulas that express each of the following properties.

(1) x is a composite number.

(2) x is a prime number.

(3) If x divides y and y divides z then x divides z

An example proof:
Let $\varphi = x \approx e \lor \exists y.\exists z.cons(y, z) \approx x$. Then

(1)	$\boxed{\varphi}$	Assumption
(2)	$\varphi(e) \approx e \approx e$	$(x = e)$
(3)	$x \approx x$	
(4)	$S(x) \approx S(x)$	EQ1
(5)	$\exists y.S(y) \approx S(x)$	$(\exists+)$
(6)	$(S(x) \approx 0) \lor \exists y.S(y) \approx S(x)$	$(\forall+)$
(7)	$\varphi \to \varphi(S(x))$	(1,6)
(8)	$\varphi[x/0] \to \varphi \to \varphi(S(x))$	(2,7)
(9)	$\forall x.\varphi(x/0) \to \varphi \to \varphi(S(x))$	$(\forall+)$

4.1 Basic Lists

Our language for lists will include a constant e, denoting the empty list, and the binary function $cons$ that creates new lists out of previous ones. $cons(a, b)$ will mean the list with a as its first element, and b as the remainder of the list. We will start with basic lists.

Definition 4.2 (basic list)**.** A list that presumes every object in the domain is a list.

In general, if we want a list containing $a_1, a_2, \ldots, a_k$ we use the object denoted by

$$cons(a_1, cons(a_2, cons(\ldots cons(a_k, e)\ldots)))$$

The values a_i can be anything in the domain; in the case of basic lists, they must of course be lists themselves. Consider the standard signature for lists

$$\sigma = \langle e, cons, \approx, \not\approx \rangle$$

We use Peano axioms for natrual numbers to form similar axioms for basic lists.

BL1 $\forall x.\forall y.cons(x, y) \not\approx e$

BL2 $\forall x.\forall y.\forall z.\forall w.(cons(x, y) \approx cons(z, w) \to (z \approx z \wedge (y \approx w)))$

BL3 For each formula $\varphi(x)$ and each variable y, $y \notin FV(\varphi)$,

$$\varphi[x/e] \to (\forall x.(\varphi \to (\forall y.\varphi[x/cons(y, x)])) \to \forall x.\varphi)$$

We adopt the following conventional notation using $\langle$ and $\rangle$:

- $\langle\rangle$ denotes the empty list, e.
- For any object a, $\langle a\rangle$ denotes the list whose single item is a, that is $cons(a, e)$ or $cons(a, \langle\rangle)$.
- For an object a and non-empty list $\langle l\rangle$, $\langle a, l\rangle$ dentoes the list whose first item is a and whose remaining items are the items on the list l. That is $\langle a, l\rangle$ denotes the list $cons(a, \langle l\rangle)$.

Exercise: Prove that every non-empty object is a `cons`, that is show that

$$\vdash_{BList} \forall x.(x \not\approx e \to \exists y.\exists z.cons(y, z) \approx x)$$

4.2 FOL Formulas for Scheme functions on basic lists

Here is the template for functions using basic lists:

```scheme
;; my-list-function: (listof any) -> any
(define (my-list-function x)
  (cond
    ((equal? x empty) ...)
    (#t ... (first x) ...
      ... (my-list-function (rest x)) ...)))
```

Example 4.3. Here is append:

```scheme
(define (Append x y)
  (cond
    ((equal? x empty) y)
    (#t (cons (first x) (Append (rest x) y)))
  )
)
```

The Scheme program `Append` uses the following objects:

- variables (`x`, `y`)
- constants (`empty`)
- relations (`equal?`)
- functions (`first, rest, cons`)
- control structures (`define, cond`)

That is,

- Scheme program $\iff$ FOL language
- variables $\iff$ FOL variables
- constants (`empty`) $\iff$ FOL constants (e)
- relations (`equal?`) $\iff$ FOL binary relations ($\approx$)
- functions $\iff$ FOL functions

Scheme functions need not to be defined for all arguments, for example `first`, and `Append`. By contrast, function symbols in FOL create new terms, which are assigned a value in each interpretation.

Example 4.4. In Scheme we have no value for `first(empty)`. If we simply use the $first(x)$ as the appropriate unary function in FOL over the domain $\mathcal{D} = \{x \mid x \text{ is a list of elements}\}$, then there exists an interpretation M in which $x^M = e$. What should be the value of $(first(x))^M$?

We need a more general approach : represent each desired partial function by an FOL relation.

An n-ary partial function $f(x_1, x_2, \ldots, x_n)$ corresponds to a $n+1$-ary relation $R_f = (x_1, x_2, \ldots, x_n, y)$ determined by the rule that $x_1, x_2, \ldots, x_n, y$ are in relation R_f iff $y = f(x_1, x_2, \ldots, x_n, y)$.

Example 4.5. Consider $\mathcal{R}_{first}$ and $\mathcal{R}_{rest}$ two binary relation symbols, where

- $\mathcal{R}_{first}(a, b)$ means "the first of a is b"
- $\mathcal{R}_{rest}(a, b)$ means "the rest of a is b"

Therefore, consider the following FOL formulas

$$\forall x.\forall y.(\mathcal{R}_{first}(x, y) \leftrightarrow \exists z.(x \approx cons(y, z)))$$

and

$$\forall x.\forall y.(\mathcal{R}_{first}(x, y) \leftrightarrow \exists z.(x \approx cons(z, y)))$$

Example 4.6. Using the list of axioms and the formulas for $\mathcal{R}_{first}$ prove that "every item except e has a first", that is give a formal prove of the FOL formula

$$\forall x.(x \not\approx e \to \exists y.\mathcal{R}_{first}(x, y))$$

We can add a relation symbol for any Scheme function.

- for a **built-in function** its deifnition in Scheme determines the appropriate FOL relation.

- for an **user-defined function**, our goal is to find formulas that characterize the appropriate relation associated to the Scheme function.

For the previous user-defined function `Append`, consider the ternary relation $\mathcal{R}_{\texttt{Append}}(x, y, z)$ which means

"the result of `(Append x y)` is `z`".

4.3 General Lists

For FOL formulas for general Scheme programs, we will find it convenient to have objects that are not lists. To have such objects, we must modify our axioms for basic lists; in particular, the induction scheme BL3 forces every object except e to be a *cons*.
Let $atom(x)$ denote the formula $\forall y.\forall z.x \not\approx cons(y, z)$.
The following are the axioms for generalized lists:

- GL1. $\forall x.\forall y.cons(x, y) \not\approx e$.
- GL2. $\forall x.\forall y.\forall z.\forall w.cons(x, y) \approx cons(z, w) \rightarrow (x \approx z \wedge y \approx w)$.
- GL3. For each formula $\varphi(x)$ and each variable y not free in φ, $\forall x.(atom(x) \rightarrow \varphi) \rightarrow (\forall x.(\varphi \rightarrow (\forall y.\varphi[x/cons(y, x)])) \rightarrow \forall x.\varphi)$.

In order to construct a FOL formula that describes the evalutation of any given Scheme program, we have to accomplish two maint asks:

- Represent a program
- Describe the execution of a program

In order to construct a FOL formula that represents a Scheme program, we must have an interpretation whose domain contains programs. First we will represent a Scheme program as a list, thus we can use the domain of lists.
Recall the scheme function `Append`.

```
(define (Append x y)
  (cond
    ((equal? x empty) y)
    (#t (cons (first x) (Append (rest x) y)))
    )
  )
```

Let's recall how we treated the parts of this program before:

- We introduced variables for the values used in the program (the arguments `x` and `y` and some intermediate values `(firstx)`, `(rest x)`, `(Append..)`)
- We used FOL constants and relations for the built-in constants, functions and relations (`empty`, `equal?`, `cons`)
- We did not directly reperesent control structures (`cond`, `define`), instead, we used `cond` to create linking of parts of FOL formulas.

For an arbitrary program, we must represent everything informally. We need the concept of a "name" of a value, function, and a way to represent names so that formulas can refer to them. Moreover, we want to be able to :

- compare names and determine whether or not they are the same

- define a "dictionary" of the meanings of names, so that we can look up a name and replace it with its meaning.

To do these, we introduce an FOL constant symbol $name$, and adopt the following convention.

A name is a list whose $first$ is the constant $name$

$$\exists x.(x \approx \langle name, y\rangle)$$

We assume a canonical way to transcribe text strings into names, and use the notations to mean the name corresponding to the strings.If s_1 and s_2 are different strings, then the corresponding names s_1 and s_2 are also different. We shall represent the keywords `cond, define`, and `lambda` by their own FOL constants, respectively $cond, define$, and λ. With these conventions, any Scheme expression can be translated into a term in the language of lists.

Example 4.7. The program `Append` becomes the term:
$\langle define, \langle Append, x, y\rangle\rangle$,
$\langle cond, \langle\langle equal?, x, e\rangle, y\rangle\rangle$,
$\langle \#t, \langle\langle first, x\rangle, \langle Append, \langle rest, x\rangle, y\rangle\rangle\rangle$

Definition 4.3 (evaluation)**. Evaluation** is the process of converting expressions to values. In Scheme the basis step of evaluation is a **substitution step** : a replacement of one part of the expression by something else. If no substitution is possible, and the expression is a value, then the expression is fully evaluated.

We use a relation **Step** to describe the substitution process. It takes three arguments:

- a list representing the current state of execution
- a list representing the dictionary of definitions of names
- a list representing a potential next state

We want the term $Step(x, D, y)$ to have the value true if and only if the expression x converts the expression y in one step, given dictionary D. For this we specify axioms, where each part of the definition of Scheme becomes one or more axiom schemata.

First we shall assume that the program never modifies a definition:

- All define statements come at the start of the program, with no two defining the same variable
- The program does not use `set!`, nor any other form of mutation

How about a program that uses local variables? Answer: Simply re-name local variables so that they all have distinct names, and then make them global.

Remark 4.1. Re-naming local variables and making them global, does not affect the program.The program may still use recursion. Simply executing the program will assign a different value to the formal argument of the recursive function at each recursive invocation. **We only forbid syntactic re-definition**.

If a name denotes a **built-in function** b, then we assume that the function is definable by a FOL relation. That is, there is a formula $\rho_b(\vec{x}, y)$ that is true iff $(b\vec{x})$ produces value y.

Example 4.8. The formula ρ_{first} for the built-in function `first` is simply $\mathcal{R}_{first}(x, y)$.

This leads to our first axiom schema for $Step$:

- Ax1. $\rho_b(\vec{x}, y) \rightarrow \mathit{Step}(\langle b, \vec{x}\rangle, D, y)$ for each built-in function b.

If a name does not have a fixed definition specified by the language, we need to look it up in the dictionary. In terms of FOL, this means that we need a relation LookUp such that $\mathit{LookUp}(x, D, y)$ evaluates to true iff the dictionary D specifies the value y for the name x. Questions: What is a dictionary? How do we specify such a relation?

Definition 4.4 (dictionary)**.** Abstractly, a dictionary is a mapping from names to values. Concretely, we shall use the standard data structure of anassociation list used in Scheme and implement it in FOL.

Definition 4.5 (association list)**.** Anassociation list is a list of pairs, where the first element of each pair is a name (key, index) and the second element is its corresponding value (or definition).

Association:

```
;; As association (as) is
;; (list k v), where
;; k is a number (the key),
;; v is a string (the value)
```

Association list:

```
;; An association list (al) is either
;; empty or
;; (cons a alst), where
;; a is an association, and
;; alst is an association list.
```

Everything else:

Proofs and Programs

Jonathan Buss

Version of October, 2013

In this module, we shall investigate proofs about complicated objects, such as computer programs. It has five sections.

Equality A look at equality in predicate logic.

Arithmetic A familiar setting, which we use to formalize inductive arguments.

Lists and Programs Extending inductive arguments to binary structures, especially those structures that occur in computer programs.

Proofs about Programs A deeper investigation, looking at entire programs as structures.

What Programs—and Logic—Can't Do A deeper investigation, looking at entire programs as structures.

1 The Equality Relation

Over any domain $\mathcal{D}$, one very useful relation is the equality relation. Formally, this relation is the set of pairs that have the same first and second element: $\{\langle x, x\rangle \mid x \in \mathcal{D}\}$. The equality relation satisfies the following axioms.

EQ1. $\forall x.\, x \approx x$ is an axiom.

EQ2. For each formula φ and variable z, $\forall x.\, \forall y.\, \big(x \approx y \to \big(\varphi[z/x] \to \varphi[z/y]\big)\big)$ is an axiom.

Axiom EQ1 states that everything is equal to itself. The schema EQ2 reflects that equal things have exactly the same properties. We can obtain further familiar properties of equality by inference.

1.1. Lemma. *The EQ axioms imply that the relation "≈" is symmetric and transitive.*[2] *That is,*

$$\vdash \forall x.\, \forall y.\, \big(x \approx y \;\to\; y \approx x\big) \quad \textit{(symmetry)}$$

and

$$\vdash \forall w.\, \forall x.\, \forall y.\, \big(x \approx y \;\to\; (y \approx w \;\to\; x \approx w)\big) \quad \textit{(transitivity).}$$

Proof. For the first, choose φ to be $z \approx x$ in Axiom EQ2, yielding $\forall x.\, \forall y.\, \big(x \approx y \to \big(x \approx x \to y \approx x\big)\big)$. Then EQ1 and modus ponens yield the required formula $\forall x.\, \forall y.\, \big(x \approx y \to y \approx x\big)$.

[1]Permission is granted to the University of Waterloo to use this material for teaching and research within the University, as specified in Policy 73 of the University.

[2]Axiom EQ1 is reflexivity; thus equality is an equivalence relation, as expected.

Symmetry:

1.	$b \approx b$	EQ1/spec [b fresh]
2.	$a \approx b \to (b \approx b \to b \approx a)$	EQ2/spec [$z \approx x$; a fresh]
3.	$\mid\ a \approx b$	Assumption
4.	$\mid\ b \approx b \to b \approx a$	$\to$-elim: 3, 2
5.	$\mid\ b \approx a$	$\to$-elim: 1, 4
6.	$a \approx b \to b \approx a$	$\to$-intro: 3–5
7.	$\forall x.\, \forall y.\, \big(x \approx y \to y \approx x\big)$	$\forall$-intro $\times$ 2: 6

Transitivity:

1.	$b \approx a \to (b \approx c \to a \approx c)$	EQ2/spec [$z = c$; a, b, c fresh]
2.	$\mid\ a \approx b$	Assumption
3.	$\mid\ a \approx b \to b \approx a$	Symmetry of $\approx$
4.	$\mid\ b \approx a$	$\to$-elim: 2,3
5.	$\mid\ b \approx c \to a \approx c$	$\to$-elim: 4,1
6.	$a \approx b \to (b \approx c \to a \approx c)$	$\to$-intro
7.	$\forall w.\, \forall x.\, \forall y.\, \big(x \approx y \to (y \approx w \to x \approx w)\big)$	$\forall$-intro $\times$ 3: 6

Figure 1: Proofs of symmetry and transitivity of equality

For the second formula, let φ be $z \approx w$ in Axiom EQ2, yielding $\forall x. \forall y. \big(x \approx y \to (x \approx w \to y \approx w)\big)$. Generalization of the free variable w then yields the required formula.

The full proofs[3] appear in Figure 1.

2 Arithmetic

The natural numbers form a basic concept of mathematics. They derive from the fundamental notion of counting things. We have a number zero for no things.[4] Once we have counted some things, if we find a next thing, we have a number to count it as well—different than all of the previous numbers. The set of all natural numbers is nothing more nor less than the collection of all numbers we can reach this way.[5]

Using the symbol "0" for the initial number, and the symbol "S" for "the next number" (or "succes-

[3]Full, that is, except for a few abbreviations:

- We do not copy axioms into the proof; we simply refer to them. In the case of $\forall$-intro applied to an axiom A, we refer to this specialization as "A/spec". In addition, we may "collapse" a use of modus ponens with the specialization step. (The referenced line determines which formula was required as an axiom.)
- We shall not use a separate step to introduce fresh variables. We shall simply indicate them in the explanation for the step. Similarly, we shall combine $\forall$-introduction steps together as one. □

[4]Originally, people counted starting with one. But mathematicians eventually realized that zero is a very useful number to have, and thus it got included.

[5]Thus, for example, more exotic things like $1/2$ or -1 are not natural numbers—we can't count up to them.

PA1. $\forall x.\, S(x) \not\approx 0$.

PA2. $\forall x.\, \forall y.\, \big(S(x) \approx S(y) \to x \approx y\big)$.

PA3. $\forall x.\, (x + 0 \approx x)$.

PA4. $\forall x.\, \forall y.\, x + S(y) \approx S(x + y)$.

PA5. $\forall x.\, x \cdot 0 \approx 0$.

PA6. $\forall x.\, \forall y.\, x \cdot S(y) \approx x \cdot y + x$.

PA7. For each variable x and formula $\varphi(x)$,

$$\varphi[x/0] \to \Big(\big(\forall x.\, \big(\varphi \to \varphi[x/S(x)]\big)\big) \to \forall x.\, \varphi\Big)$$

Figure 2: Axioms for Peano Arithmetic

sor”), we have a term for every natural number:

$$0, S(0), S(S(0)), S(S(S(0))), \ldots.$$

We augment this set of numbers by operations that have proven useful, such as addition, multiplication, etc.

The Peano axioms[6] for arithmetic are given in Figure 2. Taken together with the equality axioms above, they yield most of the known facts about the natural numbers. Axioms PA1 and PA2 give the basic structure of the natural numbers. Starting from the constant 0, there a succession of natural numbers; different numbers have different successors; and 0 is not a successor. Axioms PA3 and PA4 give a recursive (inductive) characterization of addition. Axioms PA5 and PA6 do the same for multiplication. The schema of Axiom PA7 justifies the use of recursion: the natural numbers satisfy the induction principle.[7]

These axioms imply all of the familiar properties of the natural numbers. For example, we can derive that addition is commutative.

2.1. Theorem. *Addition in Peano Arithmetic is commutative; that is, there is a proof of the formula*

$$\forall x.\, \forall y.\, x + y \approx y + x$$

from the axioms.

How can we find such a proof? Let’s take it step by step. In a formal proof, the only available properties of “+” are those given by the axioms. Combining Axioms PA3 and PA4 with Axiom PA7 (induction) will yield the required result.

[6]Named in honour of Giuseppe Peano, who first collected them together and studied their properties. The formulation here is not identical to Peano’s, but any of the equivalent axiom sets, and also similar systems of second-order logic, are called “the Peano axioms.”

[7]The formula φ appearing in the schema represents the “property” to be proved.

We need to choose a formula φ to use in Axiom PA7. To make the conclusion of PA7 be the formula we need to prove, we can choose φ to be the formula $\forall y.\, x+y \approx y+x$. With this choice of φ, we can get the required formula via modus ponens, provided that we can obtain the formulas $\varphi(0)$ and $\varphi \to \varphi[x/S(x)]$. Thus these formulas become goals of sub-proofs.

We start with the first goal, which is $\forall y.\, 0+y \approx y+0$. Since PA3 gives the value of $y+0$, namely y, we try to show that $0+y$ also has value y—that is, to derive the formula $\forall y.\, \big(0+y \approx y\big)$. How to do so? The only method available is induction: in order to complete the proof by induction, we must do a proof by induction. To help keep it all straight, let's pull out the "inner" statement and make a lemma out of it.

2.1.1. Lemma. *Peano Arithmetic has a proof of the formula* $\forall y.\, 0+y \approx y$.

Proof of lemma. To obtain the target formula via Axiom PA7, we must find proofs of the formulas $0+0\approx 0$ and $\forall y.\, \big(0+y\approx y \to 0+S(y)\approx S(y)\big)$. The former is a specialization of PA3. For the latter, PA4 gives $0+S(y)\approx S(0+y)$; also EQ2 yields $0+y\approx y \to S(0+y)\approx S(y)$.

Since $y+0\approx y$ (by PA3), we get $S(y+0)\approx S(y)$ from Axiom EQ2. PA3 also gives $S(y)\approx S(y)+0$. Thus the hypothesis $0+y$ and the axioms of equality yield $0+S(y)\approx S(y)$, as we want.

Using PA7 and modus ponens completes the required proof of the lemma.

The detailed proof:

1.	$0+0\approx 0$	PA3/spec
2.	$y+0\approx y$	PA3/spec [y fresh]
3.	$S(y+0)\approx S(y)$	EQ2/spec + MP: 2
4.	$0+S(y)\approx S(0+y)$	PA4/spec
5.	$\vert\ 0+y\approx y$	Assumption
6.	$\vert\ S(0+y)\approx S(y)$	EQ2/spec + MP: 5
7.	$\vert\ 0+S(y)\approx S(y)$	$\approx$-trans: 4, 6
8.	$0+y\approx y \to 0+S(y)\approx S(y)$	$\to$-intro: 5–7
9.	$\forall y. \big(0+y\approx y \to 0+S(y)\approx S(y)\big)$	$\forall$-intro: 8
10.	$\forall y.\, y+0\approx y$	PA7/spec + MP^2: 1, 9

□

For the second goal of the theorem, we use a second lemma.

2.1.2. Lemma. *For each free variable x,*

$$\forall y.\, x+y\approx y+x \vdash_{PA} \forall y.\, S(x)+y\approx S(x+y)\ .$$

Proof of lemma. We use induction on variable y. The basis $S(x)+0\approx S(x+0)$ we have essentially already done. For the induction step, we require $\forall z.\ (S(x)+z\approx S(x+z) \to S(x)+S(z)\approx S(x+S(z)))$; the necessary equalities come from specializing the assumption both to z and to $S(x)$. Then Axiom PA7 and modus ponens complete the proof.

In detail,

1.	$\forall y.\, x+y\approx y+x$	Premise
2.	$S(x)+0\approx 0+S(x)$	Lemma 2.1.1/spec
3.	$x+z\approx z+x$	$\forall$-elim: 1 [z fresh]

4.	$x + S(z) \approx S(z) + x$	$\forall$-elim: 1
5.	$\mid\; S(x) + z \approx z + S(x)$	Assumption
6.	$\mid\; z + S(x) \approx S(z + x)$	PA4/spec
7.	$\mid\; S(z + x) \approx S(x + z)$	EQ2/spec + $\rightarrow$-elim: 5
8.	$\mid\; S(x + z) \approx x + S(z)$	PA4/spec
9.	$\mid\; S(x) + z \approx x + S(z)$	$\approx$-elim ($\times 3$): 5, 6, 7, 8
10.	$S(x) + z \approx z + S(x) \rightarrow S(x) + S(z) \approx S(z) + S(x)$	$\rightarrow$-intro: 5–9
11.	$\forall z.\, (S(x) + z \approx z + S(x) \rightarrow S(x) + S(z) \approx S(z) + S(x))$	$\forall$-intro: 10
12.	$\forall y.\, S(x) + y \approx y + S(x)$	PA7/spec + MP^2: 2, 11

□

Whew! A lot of work, but we can now put it all together and prove the theorem.

Proof of Theorem 2.1. The proof of commutativity is a combination of the proofs in the two lemmas and a few lines to connect them and achieve the final result. Overall, the full formal proof looks as follows.

1.	$0 + 0 \approx 0$	PA3/spec
$\vdots$	... *[proof from Lemma 2.1.1]*	
12.	$\forall y.\, \big(0 + y \approx y \rightarrow 0 + S(y) \approx S(y)\big)$	
13.	$\forall y.\, 0 + y \approx y$	PA7/spec + MP^2: 1, 12
14.	$\forall y.\, 0 + y \approx y + 0$	PA3/spec + $\approx$-trans
15.	$\mid\; \forall y.\, x + y \approx y + x$	Assumption
$\vdots$	$\mid\;$ ... *[proof from Lemma 2.1.2]*	
27.	$\forall y.\, x + y \approx y + x \rightarrow \forall y.\, S(x) + y \approx S(x + y)$	$\rightarrow$-intro: 15–26
28.	$\forall x.\, \forall y.\, x + y \approx y + x$	PA7/spec + MP^2: 12, 27

This completes the proof of Theorem 2.1. □

The other familiar properties of addition, and of multiplication, have similar proofs. One can continue and define divisibility, primeness, and many other properties.

2.2. Exercise. We can express the statement, "Every non-zero natural number has a predecessor" by the formula

$$\forall x.\, \big(x \not\approx 0 \rightarrow \exists y.\, S(y) \approx x\big) \quad .$$

Show that this formula has a proof from the PA axioms.

2.3. Exercise. Write formulas that express each of the following properties.

1. (a) x is a composite number.

   (b) x is a prime number.

2. If x divides y and y divides z, then x divides z.

2.4. Exercise.

1. Prove formally that addition is associative: $x + (y + z) \approx (x + y) + z$. (Note: this is much simpler than the proof for commutativity, above. Don't let it scare you. The choice of variable for the induction does matter: if you get stuck one way, try another. [Or you can make use of commutativity.])

2. Prove that multiplication distributes over addition on the left; that is, $(x + y) \cdot z \approx x \cdot z + y \cdot z$.

3. Prove that multiplication distributes over addition on the right; that is, $x \cdot (y + z) \approx x \cdot y + x \cdot z$. Do not assume commutativity of multiplication. (If your induction seems to require commutativity of multiplication, try a different induction.)

4. Prove that multiplication is associative and commutative. Follow the corresponding proofs for addition.

3 Lists, and Programs

One can, if desired, use the natural numbers to define lists and other structures. Instead, however, we will use the Peano axioms as inspiration for axioms directly about lists. In other words, we will take lists as primitive objects and specify their properties by axioms similar to the Peano axioms. This includes induction axioms.

Our language for lists will include a constant e, denoting the empty list, and a binary function *cons* that creates new lists out of previous ones. We intend that $\mathit{cons}(a,b)$ will mean the list with a as its first element and b as the remainder of the list. To begin with, we shall consider only "basic lists", and presume that every object in the domain is a list. Later we shall allow other objects.

Basic lists

We take the following set of axioms for basic lists.

BL1. $\forall x.\, \forall y.\, \mathit{cons}(x,y) \not\approx e$.

BL2. $\forall x.\, \forall y.\, \forall z.\, \forall w.\, \mathit{cons}(x,y) \approx \mathit{cons}(z,w) \to (x \approx z \wedge y \approx w)$.

BL3. For each formula $\varphi(x)$ and each variable y not free in φ,

$$\varphi[x/e] \to \big(\forall x.\big(\varphi \to \forall y.\, \varphi[x/\mathit{cons}(y,x)]\big) \to \forall x.\, \varphi\big)$$

Axioms BL1 and BL2 correspond to the first two Peano axioms: no *cons* pair equals e, and equal *cons* pairs have equal parts. Axiom BL3 corresponds to Axiom PA7: it justifies the use of induction. Informally, it states that every object is constructed from e by the use of *cons*.

Let's look at some of the objects that must appear in a domain, in order to satisfy these axioms. Aside from e, we must have $\mathit{cons}(e,e)$ (different than e, by BL1). Thus we get a succession of objects:

$\mathit{cons}(e,e)$,
$\mathit{cons}(e,\mathit{cons}(e,e))$,
$\mathit{cons}(e,\mathit{cons}(e,\mathit{cons}(e,e)))$,
$\mathit{cons}(e,\mathit{cons}(e,\mathit{cons}(e,\mathit{cons}(e,e))))$,
$\mathit{cons}(e,\mathit{cons}(e,\mathit{cons}(e,\mathit{cons}(e,\mathit{cons}(e,e)))))$, ...

Each of these objects must be different, by BL2.

We shall regard the above objects as lists whose elements are all e. In general, if we want a list containing $a_1, a_2, \ldots, a_k$ we use the object denoted by

$$\mathit{cons}(a_1, \mathit{cons}(a_2, \mathit{cons}(\ldots \mathit{cons}(a_k, e)\ldots))) \ .$$

The values a_i can be anything in the domain; in the case of basic lists, they must of course be lists themselves.

For example, the list containing the three objects $\mathit{cons}(e,e)$, e and $\mathit{cons}(\mathit{cons}(e,e),e)$, in that order, is

$$\mathit{cons}(\mathit{cons}(e,e), \mathit{cons}(e, \mathit{cons}(\mathit{cons}(\mathit{cons}\, e,e),e))) \ .$$

When lists get large, writing out the term in full can get cumbersome. To alleviate the problem somewhat, we adopt the following conventional notation using "angle brackets."

- $\langle\,\rangle$ denotes the empty list e.
- For any object a, $\langle a\rangle$ denotes the list whose single item is a, i.e., the object $cons(a,e)$.
- For an object a and non-empty list $\langle\ell\rangle$, $\langle a,\ell\rangle$ denotes the list whose first item is a and whose remaining items are the items on the list ℓ. That is, $\langle a,\ell\rangle$ denotes the list $cons(a,\langle\ell\rangle)$.

3.1. Example.

1. Write the "angle bracket" form of the list $cons(cons(e,e),cons(cons(e,e),e))$.
2. Write the the explicit term denoted by $\langle e,e,e\rangle$.
3. Which list is longer: $\langle e,e,e\rangle$ or $\langle e,\langle e,\langle e,e\rangle\rangle\rangle$?

Answers:

1. The sub-term $cons(e,e)$ is the one-element list $\langle e\rangle$. The whole term contains that list twice; thus we denote it by $\langle\langle e\rangle\,,\langle e\rangle\rangle$
2. This is the list $cons(e,cons(e,cons(e,e)))$ that we saw above.
3. The first list is longer. It has three items, while the second has only two.

3.2. Exercise. Prove that every non-e object is a *cons*; that is, show that

$$\vdash_{BList} \forall x.\, x \not\approx e \rightarrow \exists y.\, \exists z.\ cons(y,z) \approx x \ \ .$$

(Recall Exercise 2.2, that every non-zero natural number has a predecessor.)

To make use of these axioms, we shall add other symbols to the language: constants, functions and/or relations. When we do, Axiom BL3 extends to formulas that include them.

Predicates and functions on lists

You already know a formalism for dealing with lists—Scheme programs have lists as a basic data type. Let's consider a Scheme program to append two lists, producing a third list.

```
( define (Append x y)
   ( cond ( (equal?  x empty) y )
          ( #t (cons (first x) (Append (rest x) y) ) )
   )
)
```

To use predicate logic to reason about a program such as `Append`, we must understand the basic constructs of Scheme. The code above has five kinds of objects: variables, constants, relations, functions and control structures. Variables and constants translate easily: a Scheme variable or constant (`x`, `empty`, etc.) simply becomes a variable or constant (x, e, etc.) Likewise the Scheme relation `eq` corresponds to the relation $\approx$.

A control structure such as `cond` doesn't have a single equivalent; instead, it determines the course of the computation. The Scheme language defines `cond` such that `(cond (a b) c...)` evaluates to `b` whenever `a` evaluates to `#t`. If `a` evaluates to `#f`, then the `cond` expression has the same evaluation as `(cond c...)`.

Functions pose a small problem. Scheme functions, such as `first` or `Append`, need not be defined for all arguments. By contrast, function symbols in predicate logic create new terms, which must have a value in every interpretation. Since we have no value for, say, $\mathit{first}(e)$, we cannot simply use *first* as a function symbol. There are several ways to deal with this issue. One that may look reasonable at first is to introduce a new constant symbol "error", and to use it for otherwise-undefined points. This, however, becomes very awkward: for one thing, we must then account for "error" as a value in any context. Further, programs have other ways to not produce a value than to encounter an error.

We shall therefore use explicitly the approach of representing each desired partial function by an relation. Recall that a k-ary partial function $p(x_1, x_2, \ldots, x_k)$ corresponds to the $k+1$-ary relation $R_p(x_1, x_2, \ldots, x_k, y)$ determined by the rule that $x_1, x_2, \ldots, x_k, y$ is in R_p if and only if $y = p(x_1, x_2, \ldots, x_k)$.[8]

We thus let $\mathcal{R}_{\mathtt{first}}$ and $\mathcal{R}_{\mathtt{rest}}$ be two binary relation symbols. We want "$\mathcal{R}_{\mathtt{first}}(a, b)$" to mean that "the `first` of `a` is `b`", and similarly for $\mathcal{R}_{\mathtt{rest}}$; therefore we take the axioms

$$\forall x.\, \forall y.\, \big(\mathcal{R}_{\mathtt{first}}(x, y) \leftrightarrow \exists z.\, x \approx \mathit{cons}(y, z)\big)$$

and

$$\forall x.\, \forall y.\, \big(\mathcal{R}_{\mathtt{rest}}(x, y) \leftrightarrow \exists z.\, x \approx \mathit{cons}(z, y)\big)$$

3.3. Exercise. For a relation symbol R, write sentences of predicate logic that characterize

1. whether R is the graph of a function, and
2. whether the function is total.

3.4. Exercise. Using the basic-list axioms and the ones for $\mathcal{R}_{\mathtt{first}}$, prove the following.

1. Every item except `empty` has a first; that is, $\forall x.\, \big(x \not\approx e \rightarrow \exists y.\, \mathcal{R}_{\mathtt{first}}(x, y)\big)$. (Hint: compare to Exercise 3.2.)
2. Any object x has at most one first: $\forall x.\, \forall y.\, \big((\mathcal{R}_{\mathtt{first}}(x, y) \wedge \mathcal{R}_{\mathtt{first}}(x, z)) \rightarrow y \approx z\big)$.

We can add a relation symbol for any Scheme function. For a built-in function, its definition in Scheme determines the appropriate properties. For a defined function such as `Append`, our goal is to find axioms that characterize the relation $\mathcal{R}_{\mathsf{Append}}(x, y, z)$ so that it means "the result of `(Append x y)` is `z`."

To achieve this goal, let's go through the definition of `Append`. The first line is

```
cond ( ( equal?  x empty) y )
```

The definition of `cond` gives the formula $x \approx e \rightarrow \mathcal{R}_{\mathsf{Append}}(x, y, y)$, which simplifies to

$$\mathcal{R}_{\mathsf{Append}}(e, y, y) \ .$$

For the remainder of the `cond`, we get the formula $x \not\approx e \rightarrow \varphi$, where φ is the formula from the second line:

[8]The relation R_p is called the "graph" of the function p. For the case $k = 1$, you are quite familiar with drawing a graph of a function: the set of points you mark to draw a graph of p is the relation R_p.

```
cond (#t (cons (first x) (Append (rest x) y) ) )
```

Since the formula φ must refer to the computed values of `first`, `rest` and `Append`, we use three new variables: v_f for the value of `first x`, v_r for the value of `rest x`, and v_a for the value of `(Append ...)`. The formula φ is then

$$\mathcal{R}_{first}(x, v_f) \to \mathcal{R}_{rest}(x, v_r) \to \mathcal{R}_{Append}(v_r, y, v_a) \to \mathcal{R}_{Append}\Big(x, y, cons\Big(v_f, v_a\Big)\Big)$$

Once again, our formula $x \not\approx e \to \varphi$ simplifies under the BL axioms. Since x is not `empty`, it must be a *cons* (Exercise 3.2); then Axiom BL2 yields $\mathcal{R}_{first}(x, v_f) \to \mathcal{R}_{rest}(x, v_r) \to x \approx cons\Big(v_f, v_r\Big)$. Substitution for x (Axiom EQ2) then yields the formula

$$\mathcal{R}_{Append}(v_r, y, v_a) \to \mathcal{R}_{Append}\Big(cons\Big(v_f, v_r\Big), y, cons\Big(v_f, v_a\Big)\Big) \quad .$$

In summary, the behaviour of the Scheme program `Append` is characterized by the following two formulas.

App1: $\mathcal{R}_{Append}(e, y, y)$
App2: $\mathcal{R}_{Append}(x, y, z) \to \mathcal{R}_{Append}\big(cons(w, x), y, cons(w, z)\big)$

3.5. Exercise. Prove that `Append` is total; that is, prove

$$\{\text{App1}, \text{App2}\} \vdash_{BList} \forall x.\, \forall y.\, \exists z.\, \mathcal{R}_{Append}(x, y, z)$$

3.6. Exercise.

1. Explain why the above formulas do not yield that $\mathcal{R}_{Append}$ is functional; that is,

$$\{\text{App1}, \text{App2}\} \not\vdash_{BList} \forall x.\, \forall y.\, \forall z.\, \forall w.\, \big(\mathcal{R}_{Append}(x, y, z) \wedge \mathcal{R}_{Append}(x, y, w)\big) \to w \approx z \ .$$

2. What property of the program `Append` (or indeed any Scheme program) did not get captured in the formulas above, but would allow a proof that $\mathcal{R}_{Append}$ is functional?

3.7. Exercise. Show that the function `Append` is associative; that is, for all `x`, `y` and `z`, the programs `(Append x (Append y z) )` and `(Append (Append x y) z)` produce the same result.

1. Give a formula, using the relation $\mathcal{R}_{Append}$, that states the required property.
2. Show that your formula has a proof from the list axioms and App1 and App2.

General lists

In the next section, we will find it convenient to have objects that are not lists. To have such objects, we must modify our axioms for lists; in particular, the induction scheme BL3 given above forces every object except e to be a *cons*.

Let $atom(x)$ denote the formula $\forall y.\, \forall z.\, x \not\approx cons(y, z)$. The following are the axioms for generalized lists.

GL1. $\forall x.\, \forall y.\, cons(x, y) \not\approx e$.

GL2. $\forall x.\, \forall y.\, \forall z.\, \forall w.\ cons(x, y) \approx cons(z, w) \to (x \approx z \land y \approx w)$.

GL3. For each formula $\varphi(x)$ and each variable y not free in φ,

$$\forall x.\ \big(atom(x) \to \varphi\big) \to \big(\forall x. \big(\varphi \to \forall y.\ \varphi[x/\,cons(y, x)]\big) \to \forall x.\, \varphi\big)$$

Only the induction axiom has changed from the basic version. The "base case" includes all non-lists in addition to the empty list.

3.8. EXERCISE. Re-visit Exercise 3.4 for the case of general lists. Note that you will have to modify one or both statements.

3.9. EXERCISE. Re-visit Exercises 3.5 through 3.7 for the case of general lists. For general lists, is `Append` total? Functional? Associative?

4 Formulas for general Scheme programs

In this section, we shall describe how to construct a formula of predicate logic that describes the evaluation of any given Scheme program. We have two main tasks to accomplish.

Represent a program. In order to have a formula describe any property of programs, we must have an interpretation whose domain contains programs. We shall show how to represent a Scheme program as a list; thus we can use our familiar domain of lists.

Describe the execution of a program. The definition of the Scheme language provides "substitution" rules that transform one expression into another. To evaluate a program, an interpreter applies these rules successively until no rule applies to the latest expression. We shall show how to describe this process using predicate logic.

We shall discuss each in turn.

Representing expressions and programs

We start with a look back at our example program for `Append`.

```
( define (Append x y)
    ( cond ( (equal?  x empty) y )
            ( #t (cons (first x) (Append (rest x) y) ) )
    )
)
```

You'll recall how we treated the parts of this program before.

- We used variables for the values used in the program—the arguments `x` and `y` and some intermediate values that the program does not explicitly name.
- We used constants and relations for the built-in constants, functions, and relations (`empty`, `rest`, etc.) and also for the function `Append` itself.
- We did not directly represent control elements `cond` and `define`. Instead, we used `cond` to create the formulas linking the various parts, while `define` merely indicated that we had a function to represent.

To work with arbitrary programs, however, we must represent everything uniformly. In particular, we need the concept of a "name" of a value, function or the like, and a way to represent names so that formulas can refer to them.

What do we need from names? Not much. We need to be able to compare names, to determine whether or not they are the same. Also, we need to be able to have a "dictionary" of the meanings of names, so that we can look up a name and replace it with its meaning. To do these, we introduce a constant symbol *name*, and adopt the following convention.

A name is a list whose *first* is the constant *name*.

Thus we may express "x is a name" by the formula $\exists y.\, x \approx \mathit{cons}\big(\mathit{name}, y\big)$.

We assume a canonical way to transcribe text strings into names, and use the notation $\underline{s}$ to mean the name corresponding to the string s. We require that if s and s' are different strings, then the corresponding names $\underline{s}$ and $\underline{s'}$ are also different, but we do not require other properties.[9]

We do make a few exceptions to names, however. We shall represent the keywords `cond`, `define`, and `lambda` by their own constants, respectively *cond*, *define*, and λ. Finally, `empty` and `cons` naturally become e and *cons*, respectively.

With these conventions, we can now transcribe any Scheme expression into a term in the language of lists. For example, the program above becomes the term

$$\begin{aligned}&\langle\, \mathit{define},\ \langle \underline{\mathit{Append}}, \underline{x}, \underline{y}\rangle\,,\\ &\qquad\quad \langle\, \mathit{cond},\ \langle\langle \underline{\texttt{equal?}\ }, \underline{x}, e\rangle\,, \underline{y}\rangle\,,\\ &\qquad\qquad\qquad \langle \underline{\mathit{\#t}}, \langle \mathit{cons}, \langle \underline{\mathit{first}}, \underline{x}\rangle\,, \langle \underline{\mathit{Append}}, \langle \underline{\mathit{rest}}, \underline{x}\rangle\,, \underline{y}\rangle\rangle\rangle\rangle\rangle\end{aligned}$$

Evaluation of full programs

Evaluation is the process of converting expressions to values. In Scheme, the basic step of evaluation is a substitution step: a replacement of one part of the expression by something else.[10] If no substitution is possible, and the expression is a value, then the expression is fully evaluated. The rules specifying the allowed substitutions form part of the definition of the programming language. In our translation into predicate logic, we specify these rules by giving axioms.

We use a relation *Step* to describe the substitution process. It takes three arguments: a list representing the current state of execution, a list representing the dictionary of definitions of names, and a list representing a potential next state. Thus we want the term $\mathit{Step}(x, D, y)$ to have value "true" if and only if expression x converts to expression y in one step, given dictionary D. We shall enforce this condition by specifying axioms, where each part of the definition of Scheme becomes one or more axiom schemata.

For now, we shall make some simplifying assumptions, in order to present the main ideas without getting bogged down in details. Basically, we shall assume that the program never modifies a definition. Specifically, we shall assume

- all `define` statements come at the start of the program, with no two defining the same variable, and

- the program does not use `set!`, nor any other form of mutation.

Good programming practice, as you know, violates the first assumption by using local variables. However, given any program, one can modify it into one with no local variables without changing its behaviour in any way: simply re-name local variables so that they all have distinct names, and them make them global.[11] Note that a program may still use recursion. If it does, then executing the program will

[9]If we have constants a, b, etc. for text characters, then it seems natural to take $\underline{\mathit{abc}}$ to be $\langle \mathit{name}, a, b, c\rangle$, and so on. However, we don't need this.

[10]A substitution step is sometimes called a "rewrite". These mean the same thing. Note that substitution in Scheme is essentially the same as the substitutions in formulas used in proofs.

[11]In fact, an interpreter or compiler often makes such a modification "on the fly." Two examples that you may find familiar:

- The definition of Scheme specifies that names be changed during execution. See, for example, the discussion of local definitions in Intermezzo 3 of *How to Design Programs,* by Felleisen, et al. Intermezzi 4 and 7 expand on the concept assign a different value to the formal argument of the recursive function at each recursive invocation. We do allow this; we only forbid syntactic re-definition.

The prohibition of `set!` may seem very limiting, but ultimately it turns out not to pose a problem. We shall discuss this issue later, after doing the basic translation.

Names

If a name denotes a built-in function, then we assume that the function is definable by a first-order relation. That is, there is a formula $\rho_b(\vec{x}, y)$ that is true if and only if `(b` $\vec{x}$`)` produces value y. For example, the formula $\rho_{\texttt{first}}$ for the built-in function `first` is simply $\mathcal{R}_{first}(x, y)$.

This leads to our first axiom schema for *Step*: for each built-in `b`,

Ax1: $\rho_b(\vec{x}, y) \to \mathit{Step}(\langle \underline{b}, \vec{x} \rangle, D, y)$

is an axiom.

If a name does not have a fixed definition specified by the language, we need to look it up in the dictionary. In terms of predicate logic, this means that we need a relation *LookUp* such that *LookUp*(x, D, y) evaluates to true if and only if dictionary D specifies the value y for the name x. But how do we specify such a relation? And what is a dictionary, anyway?

Abstractly, a dictionary is a mapping from names to values. Concretely, we shall use the standard data structure of an "association list"—implemented in predicate logic. You will recall that an association list is a list of pairs, where the first element of each pair is a name (or "index") and the second element is its corresponding value (or definition). If we have a dictionary and a name to look up, there are only a few possibilities.

- If the first pair in the dictionary has the given name as its first element, then the desired value is the second element of the pair. This gives the axiom

  Ax2: $\mathit{LookUp}\big(x, \langle\big\langle x, y \big\rangle, z\rangle, y\big)$

- If the first pair in the dictionary has something else as its first element, then the desired value is found by looking up the name in the rest of the dictionary. This gives the axiom

  Ax3: $x \not\approx u \to \mathit{LookUp}(x, z, y) \to \mathit{LookUp}\big(x, \mathit{cons}(\langle u, v \rangle, z), y\big)$

- If the dictionary has no first pair—it is the empty list—no name has a value in the dictionary.

Once we have *LookUp* characterized, we can use it for *Step*. We simply take the axiom

Ax4: $\mathit{LookUp}(x, D, y) \to \mathit{Step}(x, D, y)$

for expressions containing `lambda` and `set!`.

- If you have used C++, Java, or other object-oriented languages, you have likely seen names like `myType<int>(3)` or <std::basic_ostream<char, std::char_traits<char> > produced by the compiler or debugger. A programmer may write these (although need not), but they aren't the actual name used by the compiler. The actual name varies depending on the compiler and on the context in which the compiler chooses the name, but it might look something like

  `_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc`.

Taking steps

There are several possible ways to take a step in evaluating an expression. One starts by looking at the first element of the expression. If the first element is an unevaluated expression, then take a step in that expression, leaving the rest unchanged. As an axiom, we get

Ax5: $\mathit{Step}(x, D, y) \to \mathit{Step}(\mathit{cons}(x, z), D, \mathit{cons}\big(y, z\big))$.

For example, if the expression is $\langle n, x\rangle$ for a name n, then the next step is $\langle v, x\rangle$, where v is determined by either Axiom S1 (if n is a built-in) or S4 (otherwise).

There are two other cases: that the first element is itself an evaluated expression (i.e., it is a value) and that the first element of the expression is not an expression at all, but rather a control element. We shall consider the latter case next.

We exemplify control elements using `cond`. Recall the definition of `(cond (a b) c...)`: if `a` evaluates to `#f`, then `(cond (a b) c...)` has the same evaluation as `(cond c...)`, while if `a` evaluates to `#t`, then `(cond (a b) c...)` has the same evaluation as `b`. The first condition corresponds to the axiom

Ax6: $\mathit{Step}(\big\langle \mathit{cond}, \langle \underline{\texttt{\#f}}, x\rangle, y \big\rangle, D, \big\langle \mathit{cond}, y \big\rangle)$

and the second corresponds to

Ax7: $\mathit{Step}(\big\langle \mathit{cond}, \langle \underline{\texttt{\#t}}, x\rangle, y \big\rangle, D, x)$.

In the case that the guard is not a value, we evaluate it first:

Ax8: $\mathit{Step}(z, D, w) \to \mathit{Step}(\langle \mathit{cond}, \langle z, x\rangle, y\rangle, D, \langle \mathit{cond}, \langle w, x\rangle, y\rangle))$.

We now turn to the case that the first element of an expression is a value.

Values

Scheme has many kinds of values: numbers, text strings, etc. Also, functions are values. Functions differ from other values, however, in that one can apply a function to arguments (other values), producing a result. This difference affects the evaluation of a sequence of values: if `v` and `w` are values, then `(v w)` may or may not be a value. If `v` is a number, then `(v w)` is a value—a list of two values. However, if `v` is a function of one argument, then `(v w)` is not a value: one needs to apply the function in order to evaluate the expression. Values that are not functions we shall call "inert"; let $\mathit{IsInert}(x)$ denote the formula $\mathit{IsValue}(x) \wedge \neg \mathcal{R}_{\mathtt{first}}(x, \lambda)$.

- Each Scheme constant c is a value. (For example, c might be `empty`.)

  Ax9: $\mathit{IsValue}$ (c).

- A `lambda`-expression is a value:

  Ax10: $\forall x.\, \forall y.\ \mathit{IsValue}(\big\langle \lambda, \langle \mathit{name}, x\rangle, y \big\rangle)$.

- If x and y are values, and x is inert, then $\mathit{cons}\big(x, y\big)$ is a value:

  Ax11: $\mathit{IsValue}(y) \to \mathit{IsInert}(x) \to \mathit{IsValue}(\mathit{cons}\big(x, y\big))$.

Note that names are *not* values. A term of the form $\mathit{cons}(\mathit{name}, x)$ can be evaluated by looking x up in the dictionary.

Applying functions

We now reach the final case of taking a step in evaluating an expression: the application of a function. The basic definition in Scheme works by substitution: to apply a function `lambda (x) y` to a value `u`, substitute `u` for the name `x` everywhere `x` occurs in the expression `y`. This gives the axiom

$$\text{Ax12: } \mathit{IsValue}(u) \to \mathit{Subst}(\langle x,u\rangle,t,v) \to \mathit{Step}(\mathit{cons}(\langle\lambda,x,t\rangle,u),D,v))$$

The implicant $\mathit{IsValue}(u)$ appears because Scheme specifies that functions may only be applied to evaluated arguments.[12] To handle unevaluated arguments, we simply evaluate them:

$$\text{Ax13: } \mathit{Step}(u,D,v) \to \mathit{Step}(\mathit{cons}(\langle\lambda,x,t\rangle,u),D,\mathit{cons}(\langle\lambda,x,t\rangle,v))\ .$$

This leaves us with the substitution itself, which is a relatively straightforward case of inductive definition. We have the base case

$$\text{Ax14: } \mathit{Subst}(\langle x,u\rangle,e,e)$$

and the inductive case

$$\text{Ax15: } \mathit{Subst}(\langle x,u\rangle,t,v) \to \mathit{Subst}(\langle x,u\rangle,y,z) \to \mathit{Subst}(\langle x,u\rangle,\langle y,t\rangle,\langle z,v\rangle).$$

[12] This marks one significant difference between Scheme and other forms of Lisp, which allow unevaluated terms as arguments. In fact, "pure" Lisp requires that one apply a function *before* evaluating its arguments. This order of evaluation has the advantage that some programs terminate that would not terminate with the argument-first order. The results, however, can prove very surprising to programmers—especially if they use local variables.

5 What Scheme can't do

We now turn to considering some limitations of Scheme. We shall then return to predicate logic, and show that it has the same limitations.

Testing Whether a Program Halts

Some Scheme programs terminate after a finite number of steps; others do not. For example, consider the following.

```
( define (loop x) (loop loop) )
```

With this definition, the substitution rule never makes any progress[13]:

$$\begin{aligned} \texttt{(loop loop)} &\Longrightarrow \texttt{(loop loop)} \\ &\Longrightarrow \texttt{(loop loop)} \\ &\Longrightarrow \texttt{...} \end{aligned}$$

Can we distinguish between programs that halt and those that don't? Sometimes we can, of course. But can we always do it? More precisely, can we write a Scheme function `halts?` that determines whether its argument—another function—will halt on a given input? That is, we would like to have a function `halts?` that meets the following specification.

;; Contract: *halts? : SchemeProgram Input → boolean*

;; If the evaluation of `( P I )` halts, then `(halts? P I)` halts with value `#t`, and

;; If the evaluation of `( P I )` does not halt, then `(halts? P I)` halts with value `#f`.

;; Example: `( halts? loop loop )` returns `#f`.

It turns out that no such program exists. A program can do a lot towards "understanding" another program, but not everything.

5.1. THEOREM. *No Scheme function can perform the task required of* `halts?`*, correctly for all programs.*

To prove this result, we argue by contradiction. Suppose that someone claims to have a `halts?` function that meets the condition required above. By careful argument, we can show that their function fails to do the job. We won't actually analyze it directly; instead, we will write new functions that make use of it.

First, we copy their function:

```
( define ( halts? P I ) ... )
```

Next, we consider creating other functions that make use of `halts?`. For example, we can define a function that calls `halts?` with both arguments being the same function.

```
( define ( self-halt? P ) ( halts? P P ) )
```

[13]Don't try this at school—it constitutes a denial-of-service attack!

This should answer the question, “does `P` terminate when given itself as input?”

What does `self-halt?` do when given itself as input? In other words, what’s the result of the invocation `( self-halt? self-halt? )`? Let’s see.

```
( self-halt? self-halt? )
     ⟹ ( halts? ( self-halt? self-halt? ) )

     ⟹ ...    ; evaluation of halts? -- which must halt
```

$$\Longrightarrow \begin{cases} \texttt{\#t}, & \text{if } \texttt{(self-halt? self-halt?)} \text{ halts,} \\ \texttt{\#f}, & \text{if } \texttt{(self-halt? self-halt?)} \text{ doesn't halt.} \end{cases}$$

Since `halts?` always terminates, the evaluation of `( self-halt? self-halt? )` also terminates. And, since `halts?` gives the correct answer, the final result must be `#t`.

So far, so good—if a bit strange. But we can take it another step. Consider the function

```
( define ( halt-if-loops P )
  ( cond [ ( halts? P P ) ( loop loop ) ]
         [ else #t ]
  )
)
```

What happens if we invoke `halt-if-loops` with itself as its argument?

```
( halt-if-loops halt-if-loops )
    ⟹ ( cond [ ( halts? halt-if-loops halt-if-loops ) ( loop loop ) ]
              [ else #t ]
        )

    ⟹  ...    ; evaluation of halts? -- which must halt
```

$$\Longrightarrow \begin{cases} \texttt{(loop loop)}, & \text{if } \texttt{(halt-if-loops halt-if-loops)} \text{ halts,} \\ \texttt{\#t}, & \text{if } \texttt{(halt-if-loops halt-if-loops)} \text{ doesn't halt.} \end{cases}$$

The evaluation of the program `(halt-if-loops halt-if-loops)` terminates if and only if evaluation of the program `(halt-if-loops halt-if-loops)` doesn’t terminate. Impossible! No such program exists.

We made only one assumption: that the original `halts?` function worked correctly. Thus that assumption must be false: the `halts?` function we started with does not work correctly.

Thus we have proven the theorem: no Scheme function can correctly test whether a given program terminates on a given input.

Other Undecidable Problems

We define two computational problems.

PROVABILITY
Given a formula φ of predicate logic, does φ have a proof?

INTEGERROOT
Given a polynomial $q(x_1, x_2, \dots, x_n)$ with integer coefficients, does q have an integral root; that is, are there integers $a_1, a_2, \dots, a_n$ such that $q(a_1, a_2, \dots, a_n) = 0$?

5.2. THEOREM.

A. No algorithm can solve problem PROVABILITY*, correctly in all cases.*

B. No algorithm can solve problem INTEGERROOT*, correctly in all cases.*

Both proofs follow the same basic plan. We start with PROVABILITY. The proof has two steps.

1. Devise an algorithm to solve the following problem.

   Given a program `(P I)`, produce a formula $\varphi_{\mathtt{P,I}}$ such that

   $$\varphi_{\mathtt{P,I}} \text{ has a proof } \longleftrightarrow \texttt{(P I)} \text{ halts} \ .$$

2. If some algorithm solves PROVABILITY, then we can combine it with the above algorithm to get an algorithm that solves HALTING. But no algorithm solves HALTING.

Therefore, no algorithm solves PROVABILITY.

Similarly, for INTEGERROOT:

1. Devise an algorithm to solve the following problem.

   Given a program `(P I)`, produce a polynomial $q_{\mathtt{P,I}}$ such that

   $$q_{\mathtt{P,I}} \text{ has an integral root } \longleftrightarrow \texttt{(P I)} \text{ halts} \ .$$

2. If some algorithm solves INTEGERROOT, then we can combine it with the above algorithm to get an algorithm that solves HALTING. But no algorithm solves HALTING.

Therefore, no algorithm solves INTEGERROOT.

6 Undecidability and Incompleteness

We have seen that no Scheme program can, in all cases, test whether a program given as input halts on a specified input. How much "power" would some other formalism require, in order to express this halting property? Perhaps first-order logic might suffice?

> 6.1. LEMMA. *There is a Scheme program that, given a well-formed formula φ, outputs a proof of φ if one exists. If no proof exists, the program may run forever, with no output.*

PROOF SKETCH. Consider a program that generates a sequence of formulas, and then checks whether the sequence is actually a correct proof of φ. If so, it outputs the sequence. Otherwise, it starts over with another sequence.

If the program generates the sequences in a suitable order, then every possible sequence will appear eventually. Thus if any proof of φ exists, the program will eventually examine it, and then output it. □

> 6.2. THEOREM (GÖDEL'S INCOMPLETENESS THEOREM). *Let Γ be a set of formulas, such that membership in Γ is decidable; that is, there is a Scheme program that with a formula φ as input, outputs "true" if $\varphi \in \Gamma$ and outputs "false" if $\varphi \notin \Gamma$. Then there are two cases: either*
>
> 1. *$\Sigma_{GL} \cup \Gamma$ is inconsistent, or*
> 2. *There is a formula φ such that $\Sigma_{GL} \cup \Gamma \nvdash \varphi$ and $\Sigma_{GL} \cup \Gamma \nvdash \neg\varphi$.*

PROOF. Suppose that $\Sigma_{GL} \cup \Gamma$ is consistent. Consider a Scheme program that operates as follows.

```
On input S, x:
    Let $\eta$ denote the formula $\exists y.\, \mathit{Eval}(S, x, y) \wedge \mathit{final}(y)$.
    Search for a proof of either $\Sigma_{GL} \cup \Gamma \vdash \eta$ or $\Sigma_{GL} \cup \Gamma \vdash \neg\eta$.
    If a proof of $\eta$ is found, output "S halts on input x".
    If a proof of $\neg\eta$ is found, output "S does not halt on input x".
```

We argue that this program cannot halt on all formulas. If it did, it would decide the halting problem for Scheme programs—but no such program exists.

Thus there must be some S and x such that the program does not halt on inputs S and x. Therefore, the formula $\eta(S, x)$ is an example of a formula that meets the required condition—neither $\Sigma_{GL} \cup \Gamma \vdash \eta(S, x)$ nor $\Sigma_{GL} \cup \Gamma \vdash \neg\eta(S, x)$ holds. □